01168

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE INGENIERIA

DIVISIÓN DE ESTUDIOS DE POSGRADO

ANALISIS DEL PROBLEMA DEL AGENTE VIAJERO.

T E S I S

QUE PARA OBTENER EL GRADO DE:

MAESTRA EN INGENIERIA

(INVESTIGACION DE OPERACIONES)

P R E S E N T A :

ESTHER SEGURA PEREZ

DIRECTORA DE TESIS: DRA. IDALIA FLORES DE LA MOTA

CD. UNIVERSITARIA, MEXICO, D. F. NOVIEMBRE DE 2004.

A G R A D E C I M I E N T O S

No hay nada más gratificante
que la sensación de un trabajo
concluido y una meta más alcanzada.

Agradezco al Consejo Nacional de Ciencia y Tecnología (CONACyT) , por el apoyo que me proporcionó durante el período en que cursé mis estudios de Posgrado, y deseo que a estudiantes de nuevas generaciones les brinden la misma oportunidad que a mi me dieron.

Agradezco las facilidades otorgadas por el Dr. Sergio Fuentes Maya por su apoyo y sus valiosos comentarios durante mis estudios de Maestría, y de manera muy especial a la Dra. Idalia Flores de la Mota, por su comprensión y apoyo así como el valioso material que me proporcionó y sus aportaciones para la mejor conclusión de este trabajo, sin los cuales habría sido imposible el desarrollo de este trabajo.

Agradezco a mis profesores por compartirme sus conocimientos y experiencias, y muy especialmente doy las gracias a cada uno de los miembros de mi jurado por el apoyo y la confianza que me brindaron.

A la Universidad Nacional Autónoma de México, por haberme permitido ser alumno de esta noble institución y permitirme ahora, formar parte del grupo de profesionistas de este país.

Resumen

En muchos ámbitos de la industria y la tecnología la resolución de problemas complejos de optimización combinatoria es la base para el aumento de la productividad y la calidad del producto. Uno de los problemas de optimización combinatoria más conocido y estudiado a lo largo de la historia es el llamado TSP (*Traveling Salesman Problem*) o *Problema del Agente Viajero*.

El problema del TSP puede ser enunciado como: "Dado un número finito de ciudades, así como el costo asociado de viajar entre cada par de estas ciudades, encontrar el camino con menor costo asociado, de manera tal que se recorran las ciudades contempladas y se regrese a la ciudad de origen". La manera formal de enunciar el problema es: "Dado un grafo completo y pesado $G=(N,E,d)$ donde N, es el número de nodos, E es el conjunto de arcos que conectan completamente a los nodos, y d es una función que asigna un vector d_{ij} a cada arco $(i,j)\in E$, donde cada elemento corresponde a una cierta medida (costo distancia) entre i y j, entonces el problema es encontrar el circuito Hamiltoniano mínimo del grafo.

Este trabajo supone un aporte a la investigación en el sentido de que no hay documentos actualizados que brinden información completa acerca de este problema (definición, planteamiento, aplicaciones y software), es decir, hacen falta documentos en el que se nos exponga el estado actual del problema.

Es un problema que es muy estudiado, debido a la gran aplicabilidad directa que brinda, ya que el TSP surge naturalmente como un subproblema en muchas aplicaciones de logística y transporte.

Estas aplicaciones involucran la entrega de comida a domicilio, las rutas de camiones de entrega de paquetes, entre otros, lo cual redunda en beneficios a los diferentes ámbitos sociales. También existen otras áreas interesantes donde surge este problema, un ejemplo clásico es establecer el orden en que una máquina taladra agujeros en una tabla de circuitos u otros objetos, donde los agujeros serían las ciudades y el costo del viaje es el tiempo que toma la cabeza del taladro de un agujero a otro.

Por lo que es importante contar con proyectos de ésta índole para difusión e investigación de problemas que pueden aplicarse a problemas prácticos en la vida real , facilitando una mejor toma de decisión.

ÍNDICE

ANEXOS

Introducción

Indiscutiblemente el Problema del Agente viajero es quizá el problema de Optimización Combinatoria (OC) más popular de todos que puede ser resuelto con técnicas exactas de búsqueda exhaustiva o con técnicas aproximadas, como los algoritmos de construcción o de búsqueda local. A nivel mundial es un problema que ha sido objeto de estudio por investigadores, académicos, estudiantes etc, lo que no ocurre a nivel país, ya que es un tema que no ha sido lo suficientemente abordado a pesar de que es un problema tan popular y utilizado como analogía para realizar otros estudios, aunque no siempre son problemas puros del Agente Viajero, pero pueden atacarse por medio de variantes de los métodos conocidos para encontrar una solución factible.

1. Problemática

Con base en lo anterior es conveniente resaltar la importancia de la investigación de temas que son frecuentemente estudiados en otros países, dada la gran importancia que implica en los campos de la investigación y en el de la aplicación. Pues bien, con respecto al problema del agente viajero o traveling salesman problem, como se conoce a nivel mundial, no ha sido lo suficientemente investigado por lo que el campo de la aplicación también se rezaga, y los documentos que se han escrito respecto de este tema se ha quedado en el campo de la teoría, sin la aplicación práctica. Además de que son documentos que se han vuelto obsoletos, por el gran avance de investigación que se tienen en otros países con respecto a México. El documento más actual es un estudio del genoma humano que se desarrolló en el año 2001, en donde aprovechan la estructura del problema del agente viajero y las técnicas de solución que han surgido. Este documento fue elaborado por la Universidad de las Américas Puebla. El Problema del Agente Viajero es un problema combinatorio y una de las características de estos problemas es la facilidad con la que se enuncia, pero existe una dificultad inherente de resolverlo y ha sido razón suficiente para emocionar a los investigadores a diseñar algoritmos de resolución, aunque hasta la fecha no se conoce un algoritmo de solución eficiente. Es por esto que creo conveniente la realización de un documento escrito en el qué se explique en que consiste este problema y la importancia de la investigación de este problema.

2. Objetivo

Realizar un documento escrito en el que se analice el problema del agente viajero: las bases teóricas, descripción y comparación de diversos métodos de solución; así mismo, actualizar la información que existe referente a este problema, software, diseño de nuevos algoritmos de solución, y algunas aplicaciones con respecto a problemas reales.

3. Hipótesis

La elaboración de una guía escrita del problema del Agente Viajero ayudará al alumno, a entender y analizar este problema de manera práctica y sencilla, además comprenderá las características básicas de los problemas combinatorios y de redes, y la clasificación de los problemas según la teoría de la complejidad computacional, como son los problemas NP-Completos, a los cuales pertenece el Problema del Agente Viajero (PAV).

4. Justificación

En este sentido existe una necesidad de crear una guía escrita, en la que el alumno pueda entender en mayor medida el problema del agente viajero, las distintas estructuras con las que cuenta y la clasificación que se le ha dado según la teoría de la complejidad computacional en especia, desde la teoría de la NP-Completez, realizar el estado del arte en el que se actualice la información referente a este problema como: software, métodos de solución, y el diseño de nuevas estructuras del problema del agente viajero, y las aplicaciones referentes a problemas reales.

El desarrollo del trabajo está estructurado de la siguiente manera: en el capítulo uno se define el problema del agente; sin embargo, dado que la solución de este problema nos permite una doble interpretación como una permutación y como un circuito hamiltoniano, se plantean las bases teóricas de teoría de gráficas, así como la estructura de un problema combinatorio y como se trata problema NP-Completo, se definen los conceptos básicos de la teoría de la NP-Completez.

En el capítulo dos se describen algunas de las técnicas exactas que se han desarrollado para resolver problemas de optimización combinatoria. Y en especial problemas de programación lineal entera , que es una de las forma en la que se plantea el problema del agente viajero. Así, primero doy una descripción de la manera en que trabajan los algoritmos exactos, y enseguida describo el método de ramificación y acotamiento y algunas de sus variantes y también realizo una descripción del método de programación dinámica, ambos acompañados de algunos ejemplos.

En el capítulo tres se describen las principales técnicas heurísticas de solución de problemas combinatorios entre los que figuran el problema del agente viajero. Comienzo por describir los fundamentos de los algoritmos heurísticos y, enseguida, realizo una clasificación de algunas de las técnicas heurísticas y metaheurísticas que son las más utilizadas y reconocidas en la optimización combinatoria. Cabe mencionar que las técnicas heurísticas están exclusivamente diseñadas para un problema en específico con características muy especiales, mientras que las técnicas metaheurísticas son técnicas que ayudan a resolver una clase de problemas y, en este sentido, estas técnicas también se tienen que adecuar al problema que se quiere resolver. Es preciso mencionar que ha habido un gran aumento en el desarrollo de las técnicas heurísticas dadas las bondades que nos ofrecen, una buena solución y en un corto período de tiempo.

En el capítulo cuatro propongo una metodología para ayudar a resolver el problema del agente viajero. Considero software que se ha desarrollado para resolver este problema y expongo unos estudios comparativos de la eficiencia de los algoritmos heurísticos y exactos que se han realizado. Con la finalidad de tener una idea de que algoritmos se aplican más y en que condiciones, y saber qué algoritmos usar y a qué software remitirse; asimismo, tiempo actualizo la información referente al desarrollo de nuevos algoritmos en su mayoría heurísticos, y la complejidad asociada y el número de nodos que son soportados por éstos. También abordo algunas aplicaciones que toman como base este problema. Finalmente propongo las conclusiones.

Al final del trabajo inserto un anexo acerca de las etapas de solución por las que un problema combinatorio atraviesa y, otro que contiene fundamentos de teoría de gráficas.

Capítulo 1. Enfoques del Problema del Agente Viajero

El objetivo de este capítulo es plantear y definir al Problema del Agente Viajero (PAV) ó Traveling Salesman Problem (TSP) como se conoce a nivel mundial, a través de dos áreas muy definidas: en la investigación de Operaciones (IO) y, de manera especial, en la Optimización Combinatoria (OC) y dentro de la teoría de la complejidad computacional específicamente en la teoría de la NP-Completez. Y por otro lado, dado que la teoría de gráficas es una herramienta que nos ayuda a plantear y resolver problemas de una determinada complejidad a través de estructuras matemáticas cómo lo son los grafos se plantean conceptos básicos de esta teoría para plantear el problema del agente viajero, como un grafo.

Introducción

El problema del agente viajero (PAV), es un problema que se estudia en investigación de operaciones y de manera muy especial en la optimización combinatoria, cuyo conjunto de soluciones posibles es finito, pero muy numeroso para ser manejado en forma directa. Además reviste una importancia teórica para la Teoría de Complejidad, pues pertenece a la clase de los problemas combinatorios NP-Completos, para los cuales se conjetura que el tiempo de cómputo requerido para hallar la solución exacta crece al menos exponencialmente con el tamaño de la instancia considerada. Por esto es necesario buscar heurísticas que encuentren rápidamente *tours* (caminos cerrados) cercanos al óptimo. Otras de las áreas en las que se puede observar el problema del agente viajero es la teoría de gráficas, ya que es una herramienta que nos ayuda a plantear y resolver problemas como grafos. En la Figura 1.1 se muestra de manera gráfica, las dos grandes áreas o enfoques que han tomado este problema para su estudio y la teoría de gráficas que es una gran herramienta para ayudarnos a resolver problemas como lo es el PAV.

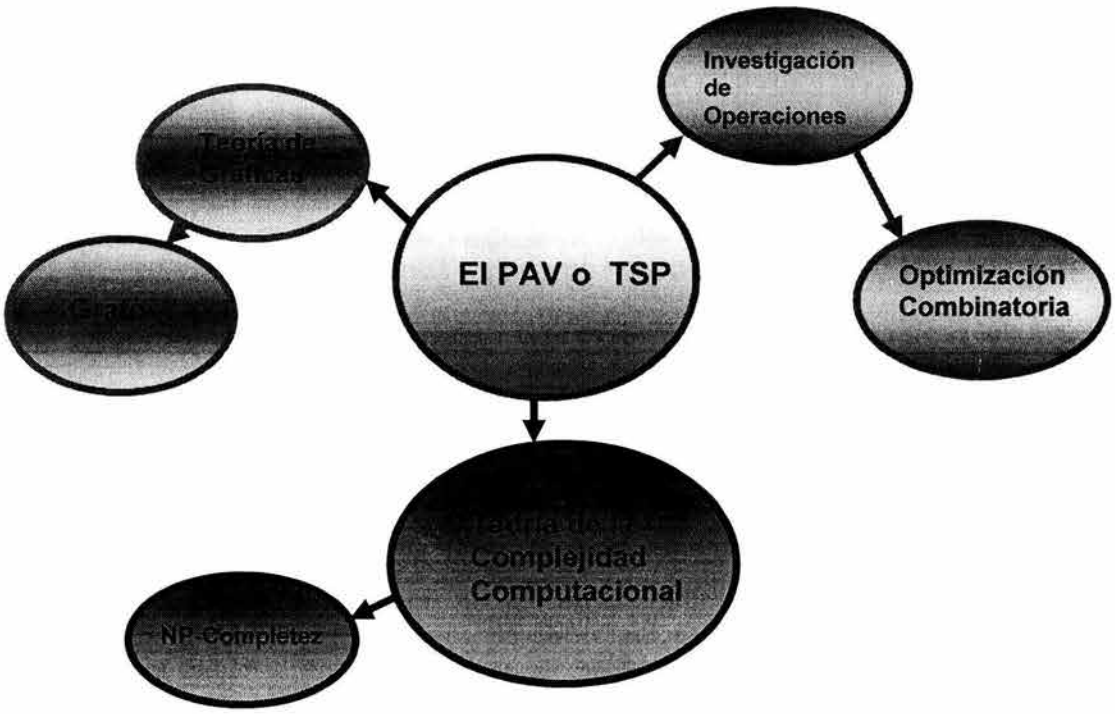


Figura 1.1 Enfoquesl del PAV

Algunas de las razones por las cuales este problema es objeto de estudio se enuncian a continuación:

- Resulta muy intuitivo y con un enunciado muy fácil de comprender.
- Es extremadamente difícil de resolver por lo que resulta un desafío constante para los investigadores del área.
- Es uno de los que más interés ha suscitado en optimización combinatoria y sobre el que se ha publicado bastante material.
- La gran mayoría de las técnicas que han ido apareciendo en el área de la optimización combinatoria han sido probadas en él, puesto que su resolución es de gran complejidad.
- Sus soluciones admiten una doble interpretación: mediante grafos y mediante permutaciones, dos herramientas de representación muy habituales en problemas combinatorios, por lo que las ideas y estrategias empleadas son, en gran medida, generalizables a otros problemas.

1. Planteamiento del problema del agente viajero

Este problema como su nombre lo dice trata de un agente viajero que desea visitar un conjunto *n* de ciudades y que se le dan los costos, las distancias o el tiempo de viajar de una ciudad que le podemos llamar *i* a una ciudad que le podemos llamar *j*. Y tiene dos condiciones: regresar a la misma ciudad de la cual partió y no repetir ciudades, es decir, si ya visitó la ciudad 3 una vez ya no se puede volver a pasar por esa ciudad. Con el objetivo de encontrar una ruta o un camino que sea el más corto posible.

Figura 1.2 Agente viajero

La definición matemática formal del problema del agente viajero requiere de una terminología y notaciones de teoría de gráficas, las cuales se verán a continuación.

1.1 Historia del problema del Agente Viajero

Si nos remitimos al PAV como un grafo (teoría de gráficas), éste tiene su inicio alrededor del año 1850, ya que este tipo de problemas (grafos) fueron tratados por dos matemáticos británicos William Rowan Hamilton, y Thomas Penyngton Kirkman . Y es justamente a Hamilton a quien se le debe el término de circuitos hamiltonianos que justamente es la solución del problema del agente viajero desde un punto de vista gráfico, originado en un juego inventado por este matemático en 1859. Trataba sobre un viaje alrededor del mundo, el cual se representaba en forma simplificada por un dodecaedro (poliedro de 12 caras pentagonales con 20 vértices), y se requiere que se pase una sola vez por cada vértice o ciudad, usando solamente las caras del dodecaedro y se regrese al punto inicial. [Flores, 2002]

Sin embargo, en 1832 se imprimió un libro en Alemania titulado "*Der handlungsreisende,wie er sein soll und was er zu thun hat,um Aufträge zu erhalten und eines glücklichen Erfols in seinen Geschäften gewiss zu sein. Von einem alten Commisvoyageur (The traveling salesman problem, how he should do to get Commissions and to be successful in his business. By a veteran traveling salesman* ". "Qué debe hacer el agente de viajes para obtener comisiones y ser exitoso en su negocio, escrito por un agente de ventas veterano" . Aunque el libro trata de otros problemas, en el último capítulo hace énfasis en la esencia del problema de agente viajero, encontrar un camino corto dentro de varias opciones y no visitar las ciudades más de una vez. [Lawler, 1985]

Y no fue hasta 1920 cuando el matemático y economista Karl Menger lo dejara entrever y lo publicara entre sus colegas en Viena, en el cual proponía el problema del mensajero. En 1930, el problema reapareció en los círculos matemáticos de Princeton. Y la primera vez que se mencionó el término de "traveling salesman problem" en los círculos matemáticos fue en 1931-1932. En 1940, ya había sido estudiado por los estadistas Mahalanobis (1940), Jessen (1942), Gosh (1948), Marks (1948) en conexión con una aplicación en la agricultura y el matemático Merill Flood lo popularizó entre sus colegas en la corporación RAND. Eventualmente, el PAV iba ganando notoriedad como un problema prototipo de problemas duros en optimización combinatoria: examinar los posibles caminos uno por uno, era una cuestión que no estaba planteada para una instancia con gran número de ciudades para los años 40's.

Este problema se abrió camino cuando George Dantzig, Ray Fulkerson, y Selmer Johnson (1954) publicaron una descripción de un método de solución del PAV titulado "*Solutions of a large scale traveling salesman problem* " , "Soluciones para el problema del agente viajero de gran escala ", ya que ilustraban el poder de ese método para resolver una instancia de 49 ciudades, un impresionante número de ciudades para aquellos tiempos.

Seleccionaron cada uno de los 48 estados de Estados Unidos (Alaska y Hawai formaron parte de Estados Unidos hasta el año de 1959) y adicionando el estado de Washington, en este problema se definían los costos de viajar de una ciudad a otra y estaban en función de la distancia de camino entre las ciudades. Ellos resolvieron el problema con 42 ciudades quitando a Baltimore, Wilmington, Philadelphia, Newark, New York, Hartford, y Providence. Dando como resultado un viaje óptimo a través de las 42 ciudades, usando la unión de la ciudad de Washington a Boston; ya que la ruta más corta entre esas dos ciudades atraviesa las siete ciudades que se quitaron. Esta solución de 42 ciudades dio una solución para el problema con 49 ciudades.

En la Figura 1.3 se tiene la solución de manera gráfica.

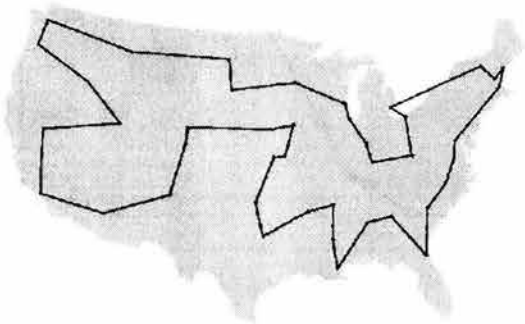


Figura 1.3 Solución del PAV de 49 ciudades en Estados Unidos

Procter and Gamble realizó un concurso en 1962. El concurso requería una solución para el PAV con 33 ciudades.

Groetchel en 1977 encontró una ruta óptima o tour óptimo de 120 ciudades en Alemania.

En 1987 Padberg y Rinaldi, encontraron una ruta óptima de 532 sucursales de AT&T en Estados Unidos. Y en ese mismo año encontraron una ruta óptima a través de una distribución de 2,392 ciudades que se obtuvo de la Incorporación Tektronics. Mientras que Groetschel y Holland encontraron una ruta óptima de 666 lugares interesantes en todo el mundo.

Hoy en día existen métodos basados en técnicas de ramificación y corte/acotamiento, las cuales explotan muy efectivamente la estructura matemática del problema, que han sido muy exitosas.

Applegate, Bixby, Chvátal, y Cook, en el año de 1994 encontraron una ruta óptima para el PAV con 7,397 ciudades que surgieron de una aplicación de los laboratorios de AT&T.

En 1998, se reportó la instancia más grande que se había resuelto de 13,509 ciudades, resuelto por Applegate, Bixby, Chvátal, y Cook lo cual evidencia el tremendo progreso logrado durante la década de los noventa.

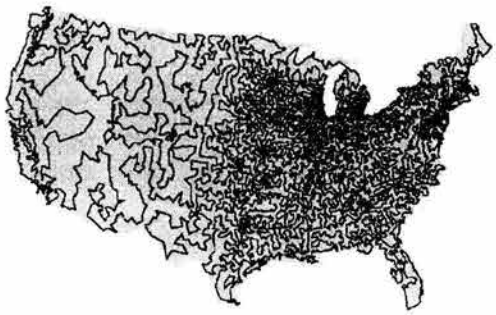


Figura 1.4 Solución óptima con 13,509 ciudades de Estados Unidos

En la Figura 1.5 se muestra de manera gráfica la relación entre el año y el tamaño de la instancia que se han ido resolviendo.

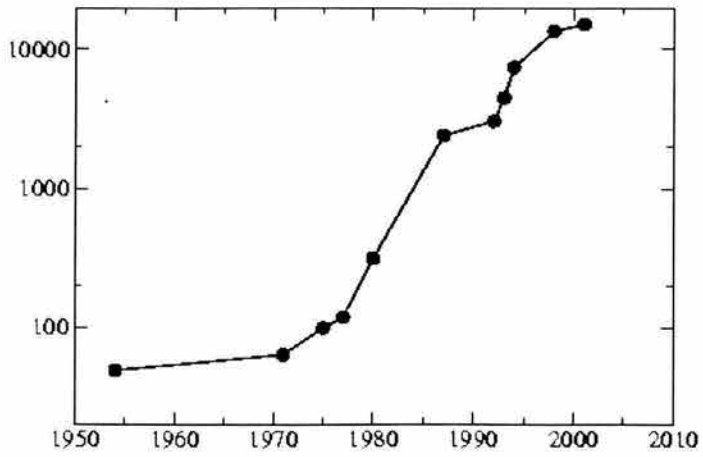


Figura 1.5 Relación gráfica entre el tamaño de instancia resuelto con respecto al tiempo

Para el año 2001 se había resuelto ya una instancia de 15,112 ciudades, localizadas en Alemania. En la figura 1.6 se tiene la solución gráfica de este problema obtenida hasta el momento. La fecha en el que se resuelve este problema es el 21 de Abril de 2001. Y lo resuelven David Applegate, R. ,Bixby,Chavátal y Cook. El nombre con el que se conoce a nivel mundial este problema es d15112. Los algoritmos usados para resolver este problema fueron planos de corte y ramificación y acotamiento. Se utilizaron 110 computadoras instaladas en paralelo en las Universidades de Rice y Princeton. El tiempo de solución computacionalmente hablando fue de 22.6 años. La solución fue llevada a cabo en una Compaq EV6 Alpha con un procesador de 500 MHz. El viaje óptimo es de 66,000 kilómetros a través de Alemania.

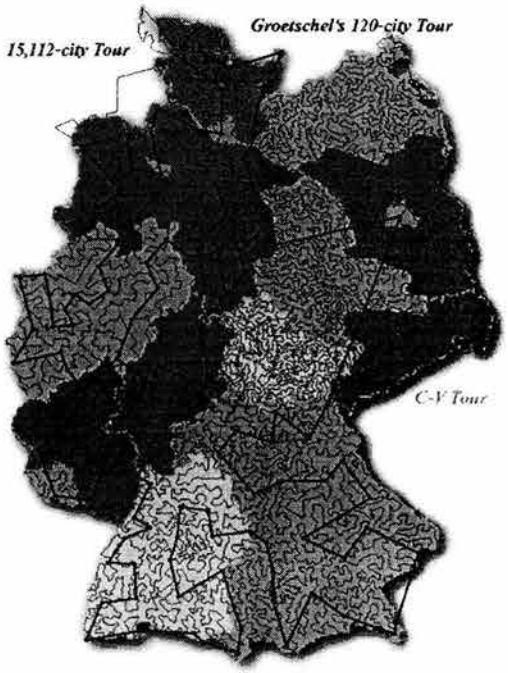


Figura 1.6 Solución óptima, gráfica del problema con 15,112 ciudades.

Actualmente se encuentran solucionando una instancia de 1, 904,711 ciudades más populares del mundo y se tiene pensado resolverse en el año 2025.

1,904,711 Cities

All Populated Cities or Towns in GEOnet Names Server

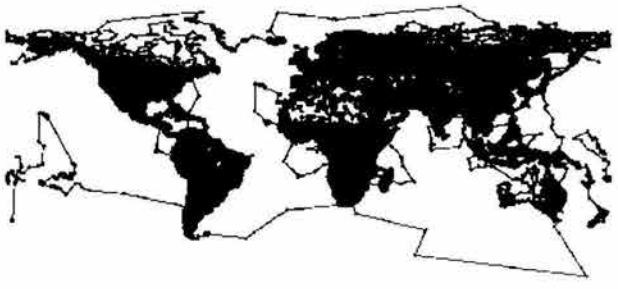


Figura 1.7 Instancia en solución de 1, 904, 711 ciudades.

La librería TSPLIB (http:// www.math.princeton.edu/tsp/usa) de dominio público contiene un conjunto de ejemplos del TSP con la mejor solución obtenida hasta el momento, y en algunos casos, con la solución óptima.

En la Tabla 1.1 se tiene un resumen de resolución del Problema del Agente Viajero, a través del tiempo, con respecto el tamaño de la instancia (número de ciudades).
En la columna de la derecha se indica el nombre del Problema del Agente Viajero como se conoce a nivel mundial.

Tabla 1.1 Resumen de resolución del PAV según tamaño de instancia

Año	**Grupo de Investigadores**	**Tamaño de la instancia**	**Nombre del PAV (TSPLIB)**
1954	G. Dantzig, R. Fulkerson, y S. Johnson	49 ciudades	dantzig42
1971	M. Held y R.M. Karp	64 ciudades	dantzig42 + 22 ciudades aleatorias
1975	P.M. Camerini, L. Fratta, y F. Maffioli	100 ciudades	hk48 + dos instancias pequeñas
1977	M. Grötschel	120 ciudades	gr120
1980	H. Crowder y M.W. Padberg	318 ciudades	lin318
1987	M. Padberg y G. Rinaldi	532 ciudades	att532
1987	M. Grötschel y O. Holland	666 ciudades	gr666
1987	M. Padberg y G. Rinaldi	2,392 ciudades	pr2392
1994	D. Applegate, R. Bixby, V. Chvátal, y W. Cook	7,397 ciudades	pla7397
1998	D. Applegate, R. Bixby, V. Chvátal, and W. Cook	13,509 ciudades	usa13509
2001	D. Applegate, R. Bixby, V. Chvátal, y W. Cook	15,112 ciudades	d15112

1.2 Teoría de gráficas y el PAV

La Teoría de Grafos se encarga de establecer los fundamentos y bases necesarias para resolver problemas de una determinada complejidad a través de estructuras matemáticas cómo lo son los grafos. Los fundamentos de esta teoría se basan en una serie de conceptos que se verán a continuación.

Una gráfica G es una pareja (V,E), donde V es un conjunto finito de *nodos* o *vértices* y los elementos de E son subconjuntos de V de cardinalidad dos, llamados *aristas*. Una gráfica se denota como $G=(V,E)$ y se representa con puntos asociados a los vértices y líneas asociadas con las aristas. Los vértices de V se denotan como $v_1,v_2,\ldots,v_k$. Una gráfica dirigida o digráfica es una gráfica con direcciones asignadas a las aristas. Es decir, una digráfica D es una pareja (V,A) donde V es un conjunto de vértices y A es un conjunto de parejas ordenadas de vértices llamados *arcos*.

La gráfica

$$G=\left(\{v_1,v_2,v_3,v_4\},\{[v_1,v_2],[v_2,v_3],[v_3,v_4],[v_4,v_1],[v_1,v_3]\}\right)$$

y la digráfica

$$D=\left(\{v_1,v_2,v_3,v_4\},\{(v_1,v_2),(v_2,v_3),(v_3,v_4),(v_4,v_1),(v_1,v_3)\}\right)$$

se muestra en la Figura 1.8.

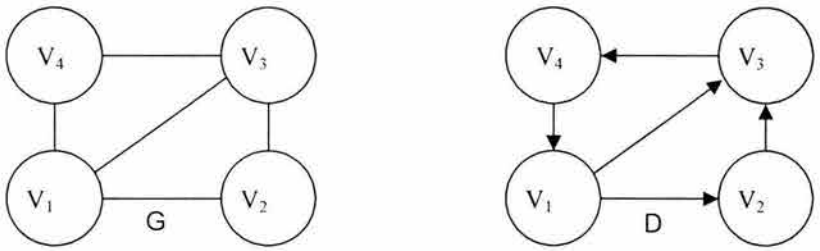


Figura 1.8 Gráfica y digráfica

Obsérvese que se usan corchetes par denotar las aristas y paréntesis para denotar los arcos.

Si $G=(V,E)$ es una gráfica y $e=[v_1,v_2]$ está en E, entonces se dice que v_1 es adyacente a v_2 (y viceversa) y e es *incidente* tanto a v_1 como a v_2. Una gráfica es *completa* si cada par de vértices son adyacentes. El *grado* de un vértice v en una gráfica G es el número de aristas incidentes a v. Por ejemplo; en la gráfica de la figura 1.9, el vértice v_3 de la gráfica G tiene grado 3.

Una gráfica parcial de $G=(V,E)$ es la gráfica $G=(V,E_p)$ donde $E_p \subset E$. Es decir, una gráfica constituida por todos los vértices y algunas aristas de G. En la figura 1.9 se muestra una gráfica G_p que es gráfica parcial de G.

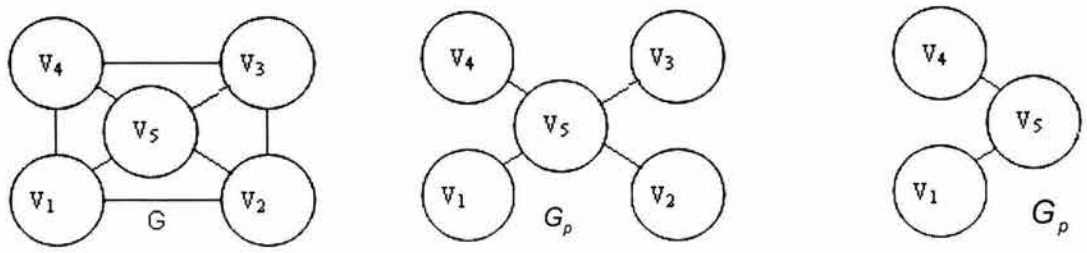


Figura 1.9 Representación de una gráfica G, gráfica parcial Gp y una subgráfica Gs

Una subgráfica de G es una gráfica $G_s = (V_s, E_s)$ donde $V_s \subset V$ y $[v_i, v_j] \in E_s$ si y sólo si $v_i, v_j \in E$. Es decir, es una gráfica formada por un subconjunto de vértices de G y todas las aristas de G que unen los vértices de este subconjunto. En la figura 1.9 G_s representa una subgráfica de G .

Una *cadena,* en una gráfica G , es una sucesión de vértices

$$W = [v_1, v_2, \ldots, v_k]$$

$k \geq 1$, tal que $[v_j, v_{j+1}] \in E$ para $j = 1 = 1, \ldots, k-1$. A W se le conoce como cadena de v_1 a v_k . Un *ciclo* es una cadena W donde $v_1 = v_k$.

Considérese la gráfica G de la figura 1.5, $[v_1], [v_4, v_3, v_1, v_2, v_3]$ y $[v_3, v_1, v_2, v_3]$ son ejemplos de cadenas; la última es un ciclo. Una gráfica es conexa si existe una cadena entre cualquier par de vértices de la gráfica.

Si la gráfica es dirigida, la sucesión de vértices W, donde $(v_j, v_{j+1}) \in A$, para $j = 1 = 1, \ldots, k-1$, se conoce como *camino* de v_1 a v_k. Un *circuito* es un camino cerrado, es decir un camino W donde $v_1 = v_k$. El camino $[v_1], [v_1, v_3, v_4, v_1]$ de la gráfica G, en la figura 1.5, es un circuito. Un *circuito hamiltoniano* de una gráfica G es un circuito que incluye todos los vértices de G.

En el contexto del problema del agente viajero a este circuito también se le conoce como *tour*. Una gráfica es hamiltoniana, si contiene un circuito hamiltoniano v_1, v_2, v_3, v_4, v_1.

Un *camino elemental* es un camino en donde no se repiten los vértices. Si los arcos del camino no se repiten se tiene un *camino simple*. Análogamente se define *cadena simple y cadena elemental*. Los conceptos tratados a continuación se aplican tanto, a gráficas como a digráficas.

Un *árbol* $T = (V_T, E_T)$ es una gráfica conexa sin ciclos. En la figura 1.10, la gráfica T constituye un árbol expandido (de expansión) de G. es decir, un árbol formado con todos los vértices de G.

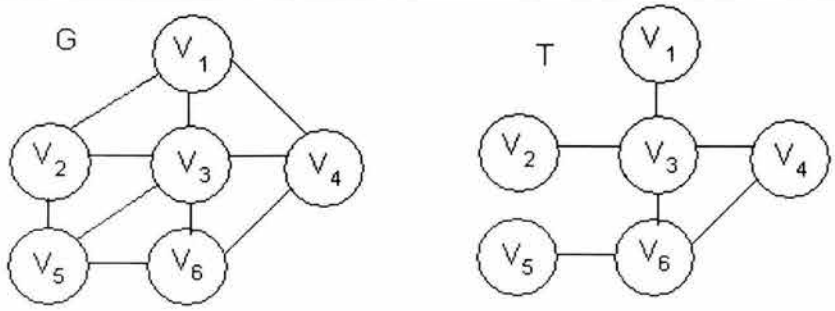


Figura 1.10 Diferencia entre grafo y árbol de expansión

Sea $T=(V_T,E_T)$ una gráfica. Las siguientes afirmaciones son equivalentes:

1. T es un árbol
2. T es conexo y tiene $|V_T|-1$ aristas
3. T no tiene ciclos, pero si se agrega una arista a T se genera un único ciclo

Una *gráfica ponderada* es una gráfica $G=(V,E)$ junto con una función de pesos W que va de E a R^+. Los pesos representan costos o distancias.

Los conceptos presentados con anterioridad permiten enunciar el problema del agente viajero, como un grafo. Supóngase que un agente viajero necesita hacer un viaje redondo a través de una colección de $p(\geq 3)$ ciudades. ¿Qué ruta debe elegir para minimizar la distancia total recorrida? Supóngase que G es una digráfica ponderada conexa cuyos vértices $(1\leq i\leq p)$ representan ciudades, y c_{ij} es el peso del arco (v_i,v_j) que representa la distancia que hay entre las ciudades v_i y v_j. El problema del agente viajero pregunta por el circuito hamiltoniano de menor peso. Si la gráfica no es dirigida, es decir, la distancia c_{ij} no depende de la dirección en la que se viaje, entonces se conoce como el problema simétrico del agente viajero. O algunas veces la gráfica no es completa, lo que significa que existen parejas de nodos que no están directamente conectados por arcos. No obstante, toda gráfica completa siempre tiene circuitos de longitud n, una que no lo es puede no tener circuitos de longitud n .Por ejemplo, la gráfica en la Figura 1.11 no tiene circuitos de longitud 5, en tal caso el problema del agente viajero no tiene solución.

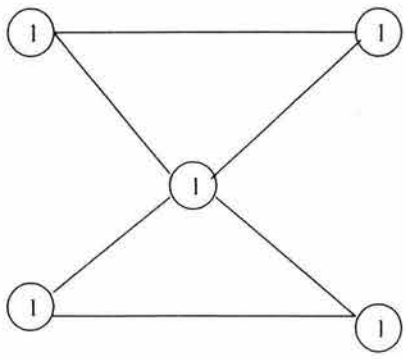


Figura 1.11 No existe solución para el problema del agente viajero

1.2.1 Circuito del agente viajero y el circuito hamiltoniano

Un circuito que incluye cada vértice de la gráfica por lo menos una vez, es llamado el circuito del viajero. Y un circuito que incluye cada vértice de la gráfica exactamente una vez, se le denomina un circuito hamiltoniano.

1.2.2 Circuito del agente viajero general

El problema del agente viajero general es el problema de encontrar un circuito hamiltoniano con una duración total mínima.

Entonces tenemos que un circuito del agente viajero con una duración mínima se le conoce como un circuito del viajero óptimo y es una solución óptima para el problema del viajero general. Un circuito hamiltoniano con una duración total mínima es llamado circuito hamiltonaino óptimo y es una solución óptima para el problema del agente viajero. Y un circuito del viajero óptimo no necesariamente es un circuito hamiltoniano óptimo. Por ejemplo, considere la gráfica que se muestra en la Figura 1.12. El único circuito hamiltoniano en esta gráfica es $(a,b),(b,c),(c,a)$ el cual tiene una duración total igual a $1+20+1=22$ unidades. El circuito del viajero óptimo $(a,b),(b,a),(a,c),(c,a)$ que pasa a través de los vértices doblemente tiene una duración total igual a $1+1+1+1=4$ unidades.

Esto es, un circuito del viajero óptimo no necesariamente es un circuito hamiltoniano óptimo.

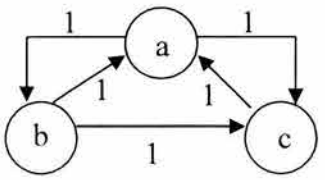


Figura 1.12 Ruta óptima del agente viajero

¿Cuándo un circuito hamiltoniano es la solución al problema del agente viajero general?

Teorema 1.1 Si para cada par de vértices x, y en la gráfica G,

$$a(x,y) \leq a(x,z) + a(z,y) \quad \forall \quad z \neq x, z \neq y \qquad (1)$$

entonces un circuito hamiltoniano es una solución óptima (si la solución existe) al problema del agente viajero general para la gráfica G.

Condición (1): La distancia que hay de x a y no es nunca más que la distancia vía cualquier otro vértice z. La condición (1) es llamada la *desigualdad del triángulo.*

Del **teorema 1.1** se tiene que si la gráfica G satisface la desigualdad del triángulo, entonces las soluciones óptimas para el problema del agente viajero de la gráfica G son soluciones óptimas del problema del agente viajero general de la gráfica G.

1.2.3 Existencia de un circuito hamiltoniano

Como se comentó en párrafos anteriores el problema del agente viajero se resuelve encontrando un circuito hamiltoniano óptimo. Desafortunadamente no todas las gráficas y consecuentemente, antes de proceder a encontrar un circuito hamiltoniano, se debería de tratar de establecer si la gráfica posee cualquier circuito hamiltoniano. En este apartado se describen varias condiciones bajo las cuales una gráfica posee un circuito hamiltoniano.

Una gráfica es llamada fuertemente conectada si para cualquier par de vértices x y y en la gráfica siempre hay una ruta o un camino. Un subconjunto de vértices X_i es llamado subconjunto de vértices fuertemente conectado, si para cualquier par de vértices $x \in X_i$ y $y \in X_i$, hay una ruta de x a y en la gráfica y X_i está contenida en otro conjunto con la misma propiedad. La subgráfica generada por el subconjunto e vértices fuertemente conectada es llamada una componente fuertemente conectada de la gráfica original.

Por ejemplo, la gráfica de la Figura1.12 está fuertemente conectada, ya que hay una ruta de un vértice a cualquier otro vértice. Ahora considere la gráfica de la Figura1.13, esta gráfica no está fuertemente conectada porque no hay una ruta del vértice *a* al vértice *b*, aunque hay una cadena del vértice b al vértice d, llamado arco (b,d). Los vértices $\{a,b,c\}$ forman una subgráfica fuertemente conectada, ya que hay una ruta de cualquier vértice a cualquier otro vértice. Más aún, no puede ser agregado otro vértice a este conjunto sin perder esta propiedad. Por ejemplo, el vértice d no puede ser agregado al conjunto, ya que no hay una ruta del vértice d al vértice a. La subgráfica generada por $\{a,b,c\}$ se muestra en la Figura 1.12.

Esta subgráfica es una componente fuertemente conectada de la gráfica original. Hay una ruta del vértice d al vértice e y una ruta del vértice e al vértice d. Sin embargo, $\{d,e\}$ no es un subconjunto de vértices fuertemente conectado porque el vértice f puede ser agregado a este conjunto sin perder la propiedad de "fuertemente conectada". No se pueden agregar otros vértices sin perder esta propiedad. Ya que $\{d,e,f\}$ es un subconjunto de vértices fuertemente conectados. La componente fuertemente conectada generada por $\{d,e,f\}$ también se muestra es la Figura 1.13

Si la gráfica *G* no es fuertemente conectada, la gráfica *G* no contiene un circuito hamiltoniano. Ya que un circuito hamiltoniano contiene una ruta entre cada par de vértices en la gráfica. Así, una condición necesaria para la existencia de un circuito hamiltoniano es que la gráfica *G* este fuertemente conectada.

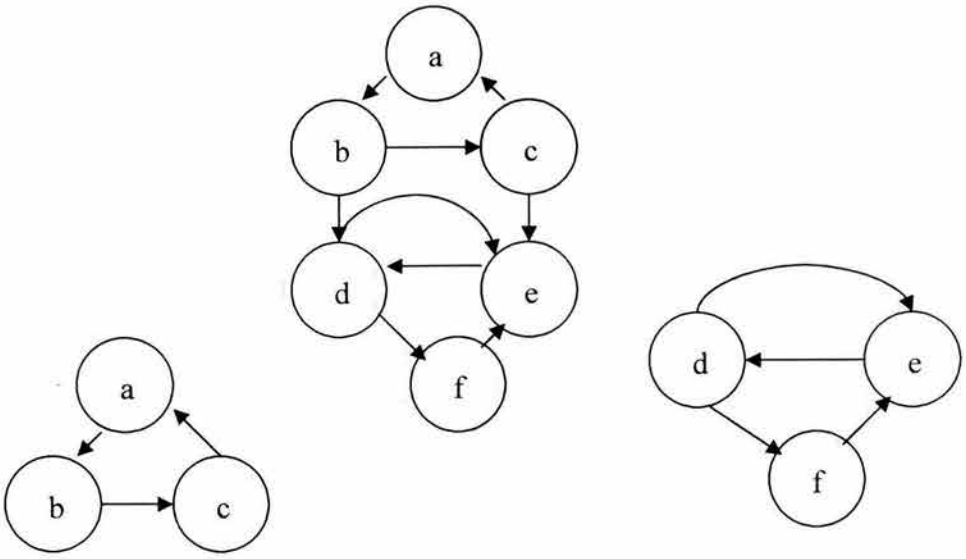


Figura 1.13 Gráfica y componentes fuertemente conectadas

1.3 Definición matemática del problema del agente viajero como un grafo

Teniendo antecedentes en los conceptos de grafos, ahora si se puede enunciar de manera formal matemática al problema del agente viajero.

Sea $G=(V,E)$ una gráfica (dirigida o no dirigida) y F una familia de circuitos hamiltonianos (*tours*) en G. Entonces el problema del agente viajero consiste en encontrar un tour (circuito hamiltoniano) en G tal que la suma de los costos de las aristas del tour sea lo mas corto posible.
Sin perder generalidad, se asume que la gráfica G es una gráfica completa (digráfica), de otra manera se podría reemplazar las aristas faltantes con aristas con costos muy altos. Sea $V=\{1,2,...,n\}$ y $C=\left(c_{ij}\right)_{nxn}$ la matriz de costos (también se le puede llamar la matriz de distancias o de pesos), donde (i,j) es la *-ésima* entrada de c_{ij} que corresponde al costo de ir del nodo *i* al nodo *j* en *G*.

El problema del agente viajero se puede ver como un problema de permutación. Y se usan generalmente dos variaciones al problema de permutación para resolver el problema del agente viajero, la *representación cuadrática* y la *representación lineal.*

Sea P_n la colección de todas las permutaciones del conjunto $\{1,2,...,n\}$. Entonces el problema del agente viajero se reduce a encontrar a $\pi=(\pi(1),\pi(2),...,\pi(n))$ en P_n tal que $c_{\pi(n)\pi(1)}+\sum_{i=1}^{n-1}c_{\pi(i)\pi(i+1)}$ sea mínimo. En este caso $(\pi(1),\pi(2),...,\pi(n))$ nos da el orden en el cual el agente debe de visitar las ciudades, empezando por la ciudad $\pi(1)$. Por ejemplo, si $n=5$ y $\pi=(2,1,5,3,4)$ entonces el tour correspondiente será $(2,1,5,3,4,2)$. En cada movimiento cíclico de π nos da el mismo tour. Entonces hay *n* permutaciones diferentes que

representan el mismo tour. Esta representación del PAV se usa generalmente es problemas de secuenciación y nos permite formularlo como un problema de programación cuadrático binario.

Por otro lado se tiene que permutaciones cíclicas son soluciones factibles. Sea P_n la colección de todas las permutaciones cíclicas de $\{1,2,...,n\}$. Entonces el PAV es encontrar $\sigma = (\sigma(1),\sigma(2),...,\sigma(n)) \in \mathbb{C}_n$ tal que $\sum_{i=1}^{n} c_{i\sigma(i)}$ sea mínimo. Bajo esta representación, $\sigma(i)$ es el sucesor de la ciudad i en el tour resultante, para $i = 1,2,...,n$. Por ejemplo si $n = 5$ y $\sigma = (\sigma(1),\sigma(2),...,\sigma(5)) = (2,4,5,3,1)$ el tour resultante está dado por: $(1,2,4,3,5,1)$. Y con esta representación se nos permite formular al PAV como un problema de permutación lineal.

Dependiendo de la naturaleza de la matriz de costos (equivalente a la naturaleza de *G*) el PAV se divide en dos clases. Si *C* es simétrica (G es no dirigida) entonces el PAV se llama el problema del agente viajero simétrico o PAVS. Si *C* es no necesariamente simétrica (equivalente a decir que *G* es dirigida) entonces el PAV se llama problema del agente viajero asimétrico o PAVA.

De manera interesante es posible formular el problema del agente viajero asimétrico en un simétrico doblando el número de nodos. Se considera el problema del agente viajero asimétrico como una digráfica $D = (N, A)$ con c_{ij} que representa el costo del arco (i, j), se construye una gráfica no dirigida $G^* = (V^*, E^*)$ con un conjunto de nodos $V^* = N \cup \{1^*, 2^*, ..., n^*\}$ donde $N = \{1,2,...,n\}$. Para cada arco (i, j) en D se crea una arista (i, j^*) en G^* con costo c_{ij} se introducen n aristas ficticias (i, i^*) con costo $-M$ donde M es un número muy grande. Todas las demás aristas tienen un costo M. Y se puede verificar que resolver el problema del agente viajero asimétrico sobre D es equivalente a resolver el problema del agente viajero simétrico sobre G^*.

1.3.1 El Problema del Agente Viajero en la Investigación de Operaciones

Como sabemos la investigación de operaciones es una herramienta muy poderosa que utiliza modelos matemáticos para ayudarnos a tomar mejores decisiones de una manera óptima. Y dentro de la investigación de operaciones se han planteado distintas herramientas de análisis como lo son: la programación entera, la programación dinámica, simulación, teoría de colas, teoría de redes, programación lineal, la optimización combinatoria, etc. Cada una de estas áreas que se han desarrollado dentro de la investigación de operaciones se ha especializado y a adquirido una estructura muy particular de los problemas que resuelve y han desarrollado su propia metodología en la forma de plantear y resolver problemas, pero el común denominador de todas éstas herramientas es la **programación matemática** como medio para plantear y resolver los problemas.
Existen diferentes formas de clasificar a los problemas dentro de la investigación de operaciones, una de ellas es con respecto al objetivo del problema o bien la naturaleza de los datos.
Si se clasifican los problemas de IO atendiendo al **objetivo del problema**, existen modelos de optimización cuyo objetivo es maximizar cierta cantidad (beneficio, eficiencia) o minimizar

cierta medida (costo, tiempo), quizás teniendo en cuenta una serie de limitaciones o requisitos que restringen la decisión (disponibilidad de capital, personal, material, requisitos para cumplir fechas límite, etc.)

Ejemplos célebres de modelos de optimización son: problemas de secuenciación, de localización, **problemas de rutas** (que es donde se encuentra el problema del agente viajero), y problemas de búsqueda. Por otro lado tenemos que, los problemas de optimización se dividen de manera natural en dos categorías: problemas de optimización con variables continuas y problemas de optimización con variables discretas. A estos últimos se les llama problemas de optimización combinatoria, al cual pertenece el problema del agente viajero.

Ahora bien, si se clasifican los problemas según **la naturaleza de los datos**, en donde se atiende más bien al tipo de modelo donde encaja el problema. En algunos casos tendremos que ajustar al problema con un ***modelo determinístico***, en el cual los datos importantes del mismo se suponen conocidos, pero en otros, algunos de estos datos se consideran inciertos y normalmente vienen dados por una probabilidad por lo que será necesario la utilización de un ***modelo probabilístico***. Sin embargo, existen modelos que conviene tratar como híbridos para estas dos categorías. Dado lo anterior, el problema del agente viajero se puede insertar dentro de los modelos determinísticos: Como son los modelos de Optimización Lineal, Asignación, Programación entera, problemas de redes, o dentro de la teoría de gráficas o bien dentro de los modelos híbridos: en la Programación Dinámica.

Si analizamos en donde se encuentra localizado el problema del agente viajero a partir de un problema de investigación de operaciones tenemos lo que se muestra en la Figura 1.14.

En el diagrama se puede observar que el problema del agente viajero es un problema que pertenece a clase de problemas de optimización combinatoria: cuyos parámetros pueden tomar únicamente valores discretos y sus soluciones se pueden definir bajo un conjunto finito de posibilidades y dado que los datos del problema son todos conocidos el problema se puede modelar mediante modelos determinísticos, sin dejar de lado que es un problema de optimización cuyo objetivo es minimizar distancias, tiempo o costos

1.3.2 Problemas de Optimización Combinatoria

En [Gutiérrez, 1991] se plantea que los problemas de optimización se dividen de manera natural en dos categorías: problemas de optimización con variables continuas y problemas de optimización con variables discretas. A estos últimos se les llama problemas de optimización combinatoria.

Y en [Flores, 2002] se nos menciona que un problema combinatorio es aquél que asigna valores numéricos discretos a algún conjunto finito de variables X, de tal forma que satisfaga un conjunto de restricciones y minimice o maximice alguna función objetivo.
Como se trata de encontrar la solución "mejor" u "óptima" de entre un conjunto de soluciones alternativas. Para resolver grandes problemas de este tipo podemos elegir entre dos caminos.

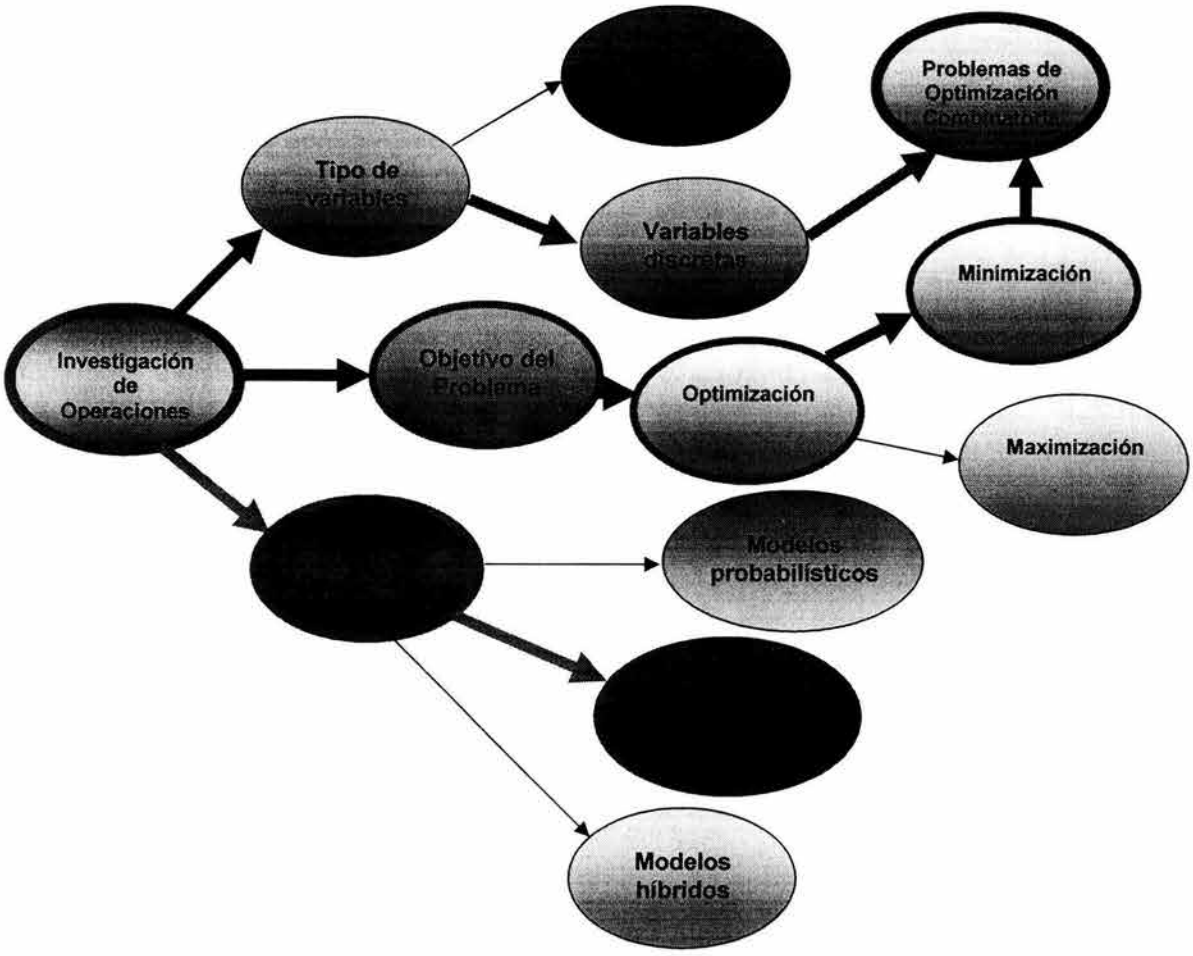


Figura 1.14 Mapa conceptual del problema del agente viajero

El primero consiste en buscar la optimalidad con el riesgo de tener grandes, posiblemente impracticables, tiempos de computación; el segundo consiste en obtener soluciones con rapidez, aun con el riesgo de caer en la sub-optimalidad. Entre los que siguen la primera opción destacan los *métodos de enumeración* y las técnicas de *programación dinámica*. La segunda opción da lugar a los *algoritmos de aproximación*, también llamados con frecuencia *algoritmos heurísticos*.

Una característica recurrente en los problemas de optimización combinatoria es el hecho de que son muy "**fáciles**" de **entender** y de enunciar, pero generalmente son "**difíciles**" de **resolver**. Podría pensarse que la solución de un problema de optimización combinatoria se restringe únicamente en buscar de manera exhaustiva el valor máximo o mínimo en un conjunto finito de posibilidades, y que usando una computadora veloz, el problema carecería de interés matemático, sin pensar por un momento en el tamaño de este conjunto.

Uno de los problemas computacionales a resolver en optimización combinatoria es la *explosión combinatoria*. La explosión combinatoria se encuentra en situaciones donde las elecciones están compuestas secuencialmente, es decir, dado un conjunto de elementos se pueden obtener diferentes arreglos ordenados de éstos, permitiendo una vasta cantidad de posibilidades. Situaciones de este tipo ocurren en problemas de inversión financiera, manejo de inventarios, diseño de circuitos integrados, manejo de recursos hidráulicos, mantenimiento de sistemas, etc. [Gutiérrez, 1991]

Los intentos por tratar con el problema de la explosión combinatoria han encontrado muchos obstáculos. Por ejemplo, no es suficiente contar con un "conocimiento experto" para manejarlos de manera efectiva; de igual manera, no es suficiente confiar en el poder computacional de alta velocidad de las super computadoras. Algunos problemas clásicos donde la explosión combinatoria prevalece, muestran que un intento por generar alternativas relevantes por computadora no es una tarea factible.

Así, por ejemplo, en el problema del agente viajero, en el cual se tiene que salir de una ciudad y regresar a la misma después de haber visitado (con costo mínimo de viaje) todas las demás ciudades, si se tienen *n* ciudades en total que recorrer entonces existen *(n-1)!* soluciones factibles, y si una computadora que pudiera ser programada para examinar soluciones a razón de un billón de soluciones por segundo; la computadora terminaría su tarea, para $n = 25$ ciudades (que es un problema pequeño para muchos casos prácticos) en alrededor de 19,674 años.

No tiene sentido resolver de esa forma un problema, si al interesado no le alcanza su vida para ver la respuesta!. [Gutiérrez, 1991]

Como se comentaba en párrafos anteriores la programación matemática es la herramienta de la cual se vale la investigación de operaciones para modelar, plantear y resolver los problemas de decisión. Pues bien la programación matemática tiene otra forma de estructurarse dentro de la Combinatoria y los pasos de solución de un problema combinatorio se pueden simplificar como: **enumeración**, **clasificación** y **optimización**. Las etapas de enumeración y clasificación se les conocen con el nombre del problema del **recuento**.

En donde la **enumeración** es la lista de los elementos que poseen cierta ó ciertas propiedades y algunas preguntas típicas son: ¿cuántas soluciones factibles diferentes existen? Desde luego, si el recuento da números demasiado elevados (y frecuentemente es el caso de la combinatoria), se renuncia a esta enumeración para realizar solamente una clasificación de los elementos mediante relaciones apropiadas, es el problema de la **clasificación**.

En algunos problemas el conjunto de las soluciones es tal que se le puede aplicar una función de valor y esta función de valor induce entonces un orden total sobre le conjunto; se pueden considerar entonces las nociones de máximo y mínimo y nos encontramos ante un problema de **Optimización** que se plantea de la forma siguiente: cuál es el sub-conjunto de soluciones para el cual la función de valor es máxima (mínima) y cuál es el valor correspondiente. En el anexo A de este trabajo se pueden ver estas etapas con un ejemplo. En realidad la forma de solucionar el problema mediante la programación matemática no difiere de la metodología de solución que utiliza la combinatoria ya que finalmente las dos formas de estructurar y resolver el problema es el mismo.
Para efectos de este trabajo se considerarán los pasos de un problema de programación matemática como medio para plantear y definir el problema del agente viajero.

1.3.3 Descripción del problema combinatorio

El enfoque clásico para estudiar un problema de optimización es proceder a identificar aquellas propiedades, cualitativas, cuantitativas, que conduzcan a uno o varios procedimientos eficientes para implementarlos en una computadora y obtener su solución.

Resulta importante aquí evaluar el tiempo que tardará un procedimiento para encontrar la solución, ya que no es lo mismo esperar unos cuantos segundos que tener que esperar horas, días o quizá más tiempo para saber la solución del problema. Otro aspecto importante es conocer el comportamiento del algoritmo cuando el tamaño del problema crece, con la finalidad de obtener un procedimiento que resulte adecuado para resolver problemas pequeños o medianos, sin embargo, dicho procedimiento resulta impracticable cuando el tamaño del problema es grande. [Gutiérrez, 1991]

La *instancia* de un problema de optimización combinatoria puede formalizarse como una pareja (S, f), donde S denota el conjunto finito de todas las soluciones posibles y f la función de costo, mapeo definido por

$$f : S \rightarrow R$$

En el caso de minimización, el problema es encontrar $i_{opt} \in S$ que satisfaga

$$f(i_{opt}) \leq f(i) \quad \forall i \in S$$

en el caso de maximización, la i_{opt} que satisfaga

$$f(i_{opt}) \geq f(i) \quad \forall i \in S$$

A la solución i_{opt} se le llama una *solución globalmente óptima* y $fi_{opt} = f(i_{opt})$ denota el costo óptimo, mientras que S_{opt} denota el conjunto de soluciones óptimas.

Un problema de *Optimización Combinatoria* es un conjunto I de instancias de un problema de optimización combinatoria.

En las definiciones anteriores se ha distinguido entre un problema y una instancia del problema. De manera informal, una instancia está dada por "los datos de entrada" y la información suficiente para obtener una solución, mientras que un problema es una colección de instancias del mismo tipo. En la sección de la teoría de la NP-Completez se explica más a detalle esta diferencia.

1.3.4 El Problema del Agente Viajero como un problema combinatorio

Considere n ciudades y una matriz (d_{pq}) de orden $n \times n$, cuyos elementos denotan la distancia entre cada par *p, q* de ciudades.

Se define un recorrido como una trayectoria cerrada que visita cada ciudad exactamente una vez. El problema es encontrar el recorrido de longitud mínima.

En este problema, una solución está dada por una permutación cíclica $\pi = (\pi(1), \pi(2), ..., \pi(n))$, donde $\pi(k)$ denota la ciudad a visitar después de la ciudad k, con $\pi^l(k) \neq k$, $l = 1, 2, ..., n-1$ y $\pi^n = k \quad \forall k$.

Aquí $\pi^l(k)$ se entiende por la aplicación de l veces la permutación π. Cada solución corresponde a un recorrido. El espacio de soluciones está dado por:

$$S = \{todas\ las\ permutaciones\ \pi\ cíclicas\ de\ las\ n\ ciudades\}$$

y la función de costo se define por

$$f(\pi) = \sum_{i=1}^{n} d_{i,\pi(i)}$$

es decir, $f(\pi)$ da la longitud del recorrido correspondiente a π. Además, se tiene que $|S| = (n-1)!$

Ahora veamos en que consiste la programación como herramienta para modelar, plantear y resolver los problemas de decisión como lo es el problema del agente viajero.

1.4 Características de un problema de programación matemática

A grandes rasgos se puede decir que un problema de programación matemática es la modelación matemática de un problema de decisión. Específicamente, los problemas de decisión suelen incluir importantes factores intangibles que no se pueden traducir directamente en términos de un modelo matemático. El principal entre estos factores es la presencia del elemento humano en casi todos y cada uno de los entornos de decisiones.

Un modelo de decisión, es sólo un medio para "resumir" un problema de decisión en una forma que permita la identificación y evaluación sistemática, de todas las opciones de decisión del problema. Así, se llega a una decisión escogiendo la opción que se considera como la "mejor" entre todas las disponibles.

Los elementos básicos de un modelo de decisión son: **opciones de decisión**, **restricciones del problema** y **criterio objetivo**, y se explicarán con el PAV.

Las **opciones de decisión** son el conjunto de todas las posibles soluciones que atañen a nuestro problema, pero éstas pueden ser factibles e infactibles. Es factible cuando cumple con las restricciones definidas de nuestro problema e infactible cuando no las cumple. Obviamente, en lo que se refiere al problema de decisión, sólo nos interesan las soluciones factibles.

Para determinar la mejor solución del problema, es necesario formular un criterio apropiado que se pueda aplicar para comparar las opciones factibles dadas. En los modelos de decisión este criterio se llama **criterio objetivo** o **función objetivo**.

Cuando evaluamos cada una de las soluciones posibles y encontramos una solución que no puede ser mejorada por ninguna otra se dice que hemos encontrado la solución óptima.

Identifiquemos estos elementos en el problema del agente viajero.
Como se comentaba en párrafos anteriores el PAV consiste básicamente en: un agente viajero debe visitar un número determinado de ciudades, con la condición de visitar cada ciudad una sola vez y regresar a la ciudad de la cual partió. Para ayudar al agente viajero a encontrar el camino que es el más corto dentro de varios caminos podríamos pensar en la siguiente solución.

1.4.1 Opciones de decisión

Para identificar las opciones de decisión o el conjunto de soluciones de nuestro problema se plantea lo siguiente. Supongamos que tenemos *n=5* ciudades. En la matriz de adyacencia simétrica que se muestra a continuación se pueden observar los costos de viajar de una ciudad *i* a otra ciudad *j*. Por ejemplo el costo de viajar de la ciudad 5 a la ciudad 2 tiene un costo de $3.

Ciudades

Ciudades

	1	2	3	4	5
1	0	3	5	7	2
2	3	0	4	6	3
3	5	4	0	7	9
4	7	6	7	0	8
5	2	3	9	8	0

Una forma de ayudar al agente viajero a encontrar el camino más corto es mostrando todas las posibles rutas como se hace a continuación:

Es importante señalar que cuando estamos armando una ruta construyamos una que sea factible, es decir, que cumpla con los requerimientos de nuestro problema los cuales son: no visitar una ciudad más de una vez y regresar a la ciudad de la cual partimos. De tal manera que la ruta 1 puede establecerse así:

Ruta posible 1: {1, 2, 3, 4, 5, 1}. Lo cual quiere decir que el agente viajero partirá de la ciudad 1, enseguida visitará la ciudad 2, después la 3, luego la 4, enseguida la 5 y por último la 1. Como podemos observar, esta solución cumple con las dos restricciones que tiene nuestro agente viajero, primera que regrese a la ciudad de la cual partió y segunda, no se repiten las ciudades, lo cual significa que no pasó más de una vez por la misma ciudad. El costo que el agente viajero tiene si elige esta ruta es de $24.
Pero el objetivo es encontrar la ruta óptima, es decir, la ruta más corta posible o bien con el menor costo, y para poder encontrar esta ruta, es necesario, mostrar todas las rutas posibles y enseguida comparar las soluciones y elegir la menor.

Ahora bien para ayudarnos a ver de manera gráfica esta primera solución nos apoyaremos en una Figura 1.15 en donde los nodos representan las ciudades y los arcos o las aristas representan el camino que une la ciudad *i* de la ciudad *j*. Esta es la primer ruta que eligió nuestro agente viajero.

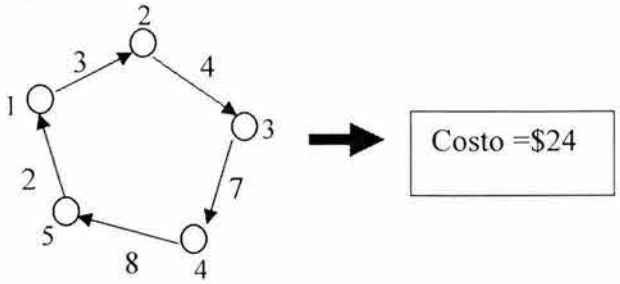


Figura 1.15 Ruta posible 1 del agente viajero

Armemos la ruta posible 2. Para armar esta ruta, de manera arbitraria, partamos de la ciudad 2 y que de esta ciudad se vaya a la ciudad 4 y después a la 5, después a la ciudad 3 y de esta a la 1 y por último que regrese a la 2. Nuevamente esta ruta es una solución factible dado que cumple con nuestras dos restricciones. Regresar a la ciudad de la cual partimos y no se repiten ciudades.
Esta ruta posible queda así: {2, 4, 5, 3, 1, 2} con un costo de 6+8+9+5+3=$31, con lo cual podemos observar que de la ruta posible 1 a la ruta posible 2 el agente viajero preferiría la ruta 1 dado que el costo asociado a esta ruta es menor en $7.

Veamos como se van construyendo las soluciones. El agente viajero en un principio tiene 5 maneras o formas de elegir su primer ciudad y una vez que se elige una ciudad la solución se reduce a *(n-1)=4* maneras de elegir a la siguiente ciudad de tal manera que tengo *n* posibilidades de elegir la primera vez, la segunda vez tengo *(n-1)* la tercera tengo *(n-2)* posibilidades, la cuarta tengo *(n-3)*, la quinta tengo *(n-4)* y ya no tengo quinta posibilidad ya que *n* es el número de ciudades que tengo de tal manera que si hubiera una quinta elección sería *(n-n=0)*. De tal manera que si queremos saber todo el conjunto de posibilidades tenemos que realizar lo siguiente: $n(n-1)(n-2)(n-3)(n-4)$ y de manera general $n(n-1)(n-2)...(n-(n-1))$ lo cual implica que el conjunto de soluciones factibles es n!. Ahora si tenemos que n=5 el conjunto de soluciones totales viene dada por n!=120 posibilidades. Imagínate construir 120 rutas posibles! Se vuelve una tarea titánica encontrar todas las soluciones factibles y después tendríamos que comparar todas las soluciones y elegir la óptima (la de menor costo), esto se vuelve una tarea extremadamente ardua.

De tal manera que todo el **conjunto de soluciones** pueden quedar de la manera siguiente:

Ruta 3= {1, 2, 4, 3, 5, 1}= $27
Ruta 4= {1, 2, 3, 5, 4, 1}= $31
Ruta 5= {1, 2, 5, 4, 3, 1}= $26
.....
Ruta n!= {5, 4, 3, 2, 1, 5}= $24

A este método de solución se le conoce enumeración exhaustiva, porque precisamente lo que se hace es mostrar todas las soluciones factibles y de ahí se elige la solución óptima, pero esto requiere de muchos cálculos y nos cuesta tiempo, dinero y esfuerzo. De ahí que se hayan descubierto formas más sencillas de resolverlo basándose en la intuición y el conocimiento empírico, pero no encontrando la solución óptima sino una buena solución.

A las técnicas de enumeración exhaustiva se les conoce como métodos de solución exacta y a los que siguen la segunda metodología son las técnicas que conocemos como métodos heurísticos, o bien metaheurísticas. En los siguientes capítulos se detallará en mayor medida éstas técnicas.

1.4.2 Restricciones del problema

Una vez que tenemos el conjunto de soluciones posibles, que en realidad se vuelve más grande con cada ciudad que le insertes, tenemos que elegir la solución que cumpla con las restricciones de nuestro problema. En este caso las restricciones son dos: **no pasar por cada ciudad más de dos veces y regresar a la ciudad de la cual partió.**

Supongamos que tenemos una ruta posible:

Ruta n_k = {1, 5, 4, 2, 5, 3, 1}

Posiblemente esta ruta ó tour sea más económica que otra, aunque cumple con la restricción de que termina la ruta en la ciudad de la cual partió, no cumple con la restricción del problema de no visitar a una ciudad más de una vez, ya que en este caso el agente viajero visita a la ciudad 5 dos veces lo cual hace que se descarte esta ruta.

O bien, podríamos tener esta otra ruta

Ruta n_{k+1} = {1, 4, 2, 5, 3, 5}

La cual viola la restricción de no regresar a la ciudad de la cual partió, y además está visitando a la ciudad 5 más de dos veces.

Entonces, cuando hablamos del conjunto de soluciones factibles, estamos hablando del conjunto de soluciones que cumplen con ambas restricciones.

1.4.3 Función o criterio objetivo

La función objetivo o criterio objetivo se establece en función de lo que queremos saber de nuestro problema. En este caso el agente viajero necesita saber cuál es la ruta más corta entre todo el conjunto de rutas factibles y que cumple con las dos restricciones.

Como se trata de un problema de minimización de rutas la función objetivo empieza con la palabra minimización.

$$\min z = \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij}$$

En donde c_{ij} es el costo asociado de viajar de la ciudad i a la ciudad j y x_{ij} es la variable de decisión de visitar la ciudad j después de visitar la ciudad i.

Antes de plantear el problema del agente viajero como un problema de programación entera binario, veamos como se plantea desde un problema de programación lineal.

1.5 El problema de programación lineal

El problema de programación matemática consiste en elegir la mejor opción de un conjunto de parámetros, en presencia de ciertas restricciones, para alcanzar un fin determinado. De manera abstracta el problema de programación lineal es el siguiente:

$$\begin{aligned} &\min(\max) f(x) \\ &sa \\ &g_i(x) \le b_i \qquad \forall i = 1,...,m \\ &h_i(x) \le c_j \qquad \forall j = 1,...,n \end{aligned}$$

Donde x es un vector de variables de decisión, y $f(x), g_i(x)$ y $h_i(x)$ son funciones lineales.

Existen muchas clases específicas de este problema, las cuales se obtienen al poner restricciones sobre las funciones bajo consideración y sobre los valores que puedan tomar las variables de decisión. En general, estos problemas se pueden dividir en dos categorías:

aquéllos con variables de decisión continuas y aquellos con variables discretas, los últimos conocidos como "problemas combinatorios". [Aguirre, 1996]

$$\min(\max)\sum_{j=1}^{n} c_j x_j$$

$$sa$$

$$\sum_{j=1}^{n} a_{ij} x_j \geq b_i \quad , \quad \forall i = 1,...,m$$

$$x_j \geq 0 \quad , \quad \forall j = 1,...,n$$

Este problema se puede escribir en forma matricial como:

$$\min cx$$

$$sa$$

$$Ax \geq b$$

$$x \geq 0$$

Donde $A \in M_{mxn}$ de entradas a_{ij} con $i = 1,....,m \;\; y \;\; j = 1,...,n.$

1.5.1 El PAV y el problema de Programación Lineal Entera Binario

Una vez que se ha explicado en que consiste la programación matemática se empezará a definir al problema del agente viajero desde un problema de programación lineal hasta un problema de programación lineal entera binario.

Dentro de la clasificación de los modelos de investigación de operaciones según la naturaleza de los datos se comentaba que los modelos pueden ser determinísticos, probabilísticas o bien híbridos. La programación lineal cae dentro de los modelos determinísticos ya que los datos del modelo se suponen conocidos, pero en muchas ocasiones las variables de decisión sólo pueden asumir valores enteros: número de aviones a comprar, número de ventanas que construir, número de viajes... sin embargo, cuando las variables solo pueden asumir valores enteros hay un número finito de posibles valores que pueden asumir (aunque puede ser una gran cantidad de valores posibles).

Los modelos de programación lineal entera pueden ser clasificados en uno de tres:

Modelo	**Tipo de variable de decisión**
Todo entero (AILP)	Todas son enteras
Entero-mixto	Algunas son enteras
Binario	Todas son 0 ó 1

Tabla 1.2 Clasificación de los modelos de programación lineal entera

En este caso el problema del agente viajero es un problema que se puede acotar en un modelo binario ya que la variable la decisión es 1 si el agente viajero visita la ciudad *j* después de visitar la ciudad *i* y será 0 en caso contrario. A estas variables de decisión se le conocen como binarias y a los problemas que se plantean con variables de decisión binarias se les llama problemas de programación lineal entera binarios. Y esta es la forma en la que se plantea el problema de la agente viajero.
A continuación se verá como se plantea de manera matemática como un problema de programación lineal entera binario.
Sea X_{ij} la variable de decisión 0-1, que nos indica si el agente viajero viaja de la ciudad i a la ciudad j, y sea c_{ij} la distancia correspondiente. Entonces la distancia de un "tour" (camino cerrado) o una ruta es: $\sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij}$

La forma de plantear de manera matemática la primera restricción que nos dice que se debe salir de cada ciudad exactamente una vez, es de la siguiente forma:

$$\sum_{j=1}^{n} x_{ij}=1 \qquad i=1,...,n$$

y de manera similar, la segunda restricción, nos dice que se debe entrar en cada ciudad exactamente una vez. Es decir, $\sum_{i=1}^{n} x_{ij}=1 \qquad j=1,...,n$

Si pensamos que además de que el agente viajero no se le permite realizar subtours, es decir, no se le permite visitar solamente un conjunto de ciudades sino todas, tendríamos una tercer restricción.

Hay distintos caminos para lograr la restricción de "no se permiten subtours". Esta restricción se puede escribir de la siguiente manera:

Sea S cualquier subconjunto propio de V o bien un conjunto no vacío de $N=\{1,...,n\}$ y $|\cdot|$ denota la cardinalidad. Si las aristas correspondientes a $x_{ij}=1$ con ambos extremos en S son menos que $|S|$ entonces no se forman subtours; es decir, a lo más debe haber $|S|-1$ aristas. Por lo tanto, para eliminar subtours se debe cumplir la siguiente desigualdad:

$$\sum_{i\in S}\sum_{j\in S} x_{ij} \le |S|-1 \qquad \forall \quad S\subset V, S\neq 0$$

O con la siguiente fórmula:

$$\sum_{i\in S}\sum_{j\in S} x_{ij} \ge 1$$

Cualquier subtour viola las dos restricciones anteriores para cualquier conjunto S. Así también representan un gran número de restricciones: $2^n/2$ para ser exactos. De acuerdo con lo anterior, el PAV se puede plantear de la siguiente manera:

$$\min Z = \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij}$$

s.a.

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad \forall i = 1,...,n \quad i \neq j$$

$$\sum_{i=1}^{n} x_{ij} = 1 \qquad \forall\ j = 1,...,n \quad i \neq j$$

$$\sum_{i\in S}\sum_{j\in S} x_{ij} \geq 1 \qquad \forall\ \mathrm{S} \subset \mathrm{V}, \mathrm{S} \neq 0$$

$$x_{ij} = 0,1 \qquad \forall\ 1 \leq i \neq j \leq n$$

Por ejemplo, considérese la gráfica de la Figura 1.16

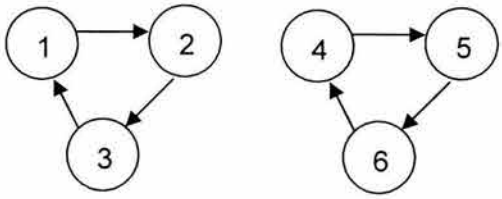


Figura 1.16 Subtours

Sea $S = \{1,2,3\}\,(o\ S = \{4,5,6\})$. El lado izquierdo de la desigualdad $\sum_{i\in S}\sum_{j\in S} x_{ij} \leq |S| - 1 \quad \forall \mathrm{S} \subset \mathrm{V}, \mathrm{S} \neq 0$ (Figura 1.16) es 3, mientras que $|S| - 1 = 2$. Por lo tanto la desigualdad no se cumple y como consecuencia se forman subtours. La formulación del problema del agente viajero en este caso es:

$$\min Z = \sum_{i\in V}\sum_{j\in V} c_{ij}x_{ij}$$

s.a.

$$\sum_{j\in V} x_{ij} = 1 \qquad i \in V$$

$$\sum_{i\in V} x_{ij} = 1 \qquad \mathrm{j} \in V$$

$$\sum_{i\in S}\sum_{j\in S} x_{ij} \leq |S| - 1 \qquad \forall \mathrm{S} \subset \mathrm{V}, \mathrm{S} \neq 0$$

$$x_{ij} = 0,1 \qquad \forall i, j \in V$$

1.5.2 El problema del Agente Viajero Simétrico (PAVS)

Si se considera el problema simétrico del agente viajero, donde $c_{ij} = c_{ji} \quad \forall i, j \in V$, la formulación es la siguiente:

$$\min Z = \sum_{i \in V} \sum_{j > i} c_{ij} x_{ij}$$

$s.a.$

$$\sum_{j<i} x_{ij} + \sum_{j<i} x_{ij} = 2 \qquad i \in V$$

$$\sum_{i \in S} \sum_{j \in S, j>i} x_{ij} \leq |S| - 1 \qquad \forall S \subset V, S \neq 0$$

$$x_{ij} = 0,1 \qquad \forall i, j \in V$$

1.5.3 El problema del Agente Viajero Asimétrico (PAVA)

Aquí las distancias de viajar de la ciudad i a la ciudad j es diferente a la de viajar de la ciudad j a la ciudad i, por lo que tendríamos una matriz de adyacencia asimétrica. La representación matemática se puede ver en el planteamiento del PAV como problema de programación lineal entera binario.

La estructura matemática que se planteó en las secciones anteriores es la estructura que tiene el problema del agente viajero clásico, sin embargo este problema se parece a otros problemas de optimización como el problema de asignación, árboles de expansión mínimos, trayectoria mas larga entre otros.

A continuación se explicarán los modelos matemáticos y la similitud con el problema del agente viajero.

1.6 Relación con otros problemas de optimización

Debido a las propiedades estudiadas en las secciones anteriores, se observa que el problema del agente viajero tiene ciertas similitudes con otros problemas de programación matemática. Auxiliándonos de la solución de estas programaciones, se puede obtener la solución del PAV de una manera más fácil. La relación con otros problemas de optimización también nos ayuda a encontrar ciertos parámetros del problema del viajero tales como: la cota inferior de la función objetivo, la probabilidad de que una solución de otros problemas sea un circuito hamiltoniano etc. Los casos más usuales del problema de programación matemática que tienen relación con el problema del viajero se describen a continuación.

1.6.1 El PAV y el problema de asignación

El problema de **asignación** es el siguiente: un conjunto de n personas están disponibles para realizar n tareas. Si la persona i realiza la tarea j, se genera un costo de c_{ij} unidades. El problema consiste en encontrar una asignación $\{\pi_1, ..., \pi_n\}$ que minimice $\sum_{i=1}^{n} c_{i\pi}$, donde

π_i es la tarea realizada por la persona i.Aquí, la solución está representada por la permutación $\{\pi_1,...,\pi_n\}$ de los números $\{1,...,n\}$.

Si se adecua el problema del agente viajero al problema de asignación se tiene lo siguiente:

Sea X_{ij} la variable de decisión 0-1, que nos indica si el agente viajero viaja de la ciudad i a la ciudad j, y sea c_{ij} la distancia correspondiente. Entonces la distancia de un "tour" o una ruta es:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij}$$

Esta primera restricción nos dice que se debe salir de cada ciudad exactamente una vez.

$$\sum_{j=1}^{n} x_{ij}=1 \qquad i=1,...,n$$

y de manera similar, la segunda restricción, nos dice que se debe entrar en cada ciudad exactamente una vez. Es decir,

$$\sum_{i=1}^{n} x_{ij}=1 \qquad j=1,...,n$$

Excepto por el requerimiento de que la solución sea un circuito hamiltoniano, la formulación es un modelo de asignación. Desafortunadamente no hay una garantía de que la solución óptima del modelo de asignación será un circuito hamiltoniano. Más bien la solución nos presentará una serie de subcircuitos a partir de los cuales podremos construir dicho circuito. Así,el problema de asignación es una relajación del PAV o de manera equivalente, el PAV es la restricción del problema de asignación, adicionando la restricción "no se permiten subtours". Es decir, no se permiten rutas que no abarcan todo el conjunto de vértices, nodos o bien de ciudades. Sin embargo se puede encontrar la probabilidad de que la solución del problema de asignación, sea un circuito hamiltoniano suponiendo que cada elemento de la matriz de costos sea una variable aleatoria.
Sea el problema de asignación modificado

$$\begin{aligned}
&\min w=\sum_{i}\sum_{j} c_{1j}x_{1j}\\
&sujeto \quad a \quad \sum_{i} x_{1j}=1 \quad \forall j=1,...,n\\
&\qquad\qquad \sum_{j} x_{1j}=1 \quad \forall j=1,...,n\\
&\qquad\qquad x_{11}=0 \quad \forall i
\end{aligned}$$

Donde c_{1j} son variables aleatorias independientes e idénticamente distribuidas.

Teorema 1.1: Supongamos que las soluciones del problema anterior tienen la misma probabilidad de que la solución óptima sea un circuito factible es:
$Pn=(n-1)!/[n!e^{-1}+0.5]\approx e/n$ para n grande.

Prueba: Primero demostraremos que el número de soluciones enteras es $[n!e^{-1}+0.5]$.

Si resolvemos el problema

$$\min w = \sum_i \sum_j c_{1j} x_{1j}$$

$$\sum_i x_{1j} = \sum_j x_{j1} = 1 \quad \forall j = 1,...,n$$

Existen exactamente $n!$ soluciones factibles. Consideramos que se obtiene del problema anterior otro problema haciendo $x_{kk} = 0$, y eliminando la columna *k* y el renglón *k*. El problema modificado obtenido tiene $(n-1)!$ soluciones, y hay $\binom{n}{1}$ maneras para escoger *k* y 1.

Ahora consideramos el problema obtenido por escoger 2 índices *k* y 1, haciendo $x_{kk} = x_{11} = 0$. De esta manera, vamos a resolver un problema de asignación de tamaño $(n-2)*(n-2)$. El número de soluciones es $(n-2)!$ y hay $\binom{n}{2}$ manera para escoger *k* y 1.

Aplicando el principio de inclusión y exclusión, se obtiene el número de soluciones del problema de asignación modificado que es:

$$n! - \binom{n}{1}(n-1)! + \binom{n}{2}(n-2)! - \binom{n}{3}(n-3)! + ... + (-1)^n \binom{n}{n}(n-n)!$$

$$= n!(1 - 1/1! + 1/2! - 1/3! + ... + (-1)^n / n!)$$

Los términos que están dentro de l paréntesis son los primeros n términos de la serie de expansión de e^{-1}. Se puede demostrar que el error de estimación de la expresión es menor que 0.5 para $n \succ 1$. Por tanto, el número de soluciones es dado por $n!e^{-1} + 0.5$ redondeado al entero más cercano, o sea $[n!e^{-1} + 0.5]$.

$$Pn = (n-1)!/[n!e^{-1} + 0.5]$$

Y esto termina la prueba.

1.6.2 El PAV y los árboles de expansión mínimos

Un árbol de expansión de una gráfica es un árbol conectado en todos los vértices. La solución al problema es encontrar un árbol de expansión con una duración mínima. El primer título que se le dio a este problema fue "*On the shortest spanning subtree of a graph and traveling salesman problem*" (Sobre subárboles de expansión mínimos de una gráfica y el problema del agente viajero). Cada ruta hamiltoniana contiene y satisface la restricción adicional "ningún vértice del árbol tiene un grado mayor a dos".

Esto es, que el problema de expansión mínimos es una relajación del PAV, y el problema del Agente Viajero es una restricción del problema de árboles de expansión mínimos.

El problema de árbol de expansión mínima esta definido como: dada la matriz de costos de una gráfica no dirigida, encontrar el árbol que incluye todos los vértices en la gráfica, tal que el costo total del árbol sea mínimo.

En una solución factible del problema del agente viajero, el grado de cualquier vértice es 2, pero la solución del árbol de expansión mínimo no cumple esta condición aunque la gráfica

esté conectada. Sin embargo, se puede obtener la solución del problema del agente viajero desde este árbol, transformándolo hasta que la cardinalidad de todos los nodos sea 2.

En el caso de una gráfica no dirigida con matriz de costo simétrica, una cota inferior para la solución del problema del agente viajero es derivada usando el correspondiente árbol de expansión mínimo de la gráfica de la siguiente manera. Supongamos que el arco (x_1, x_2) está en el circuito óptimo del problema del agente viajero. Si este arco es borrado del circuito, se genera una trayectoria de $n-1$ arcos que recorre todos los vértices empezando de x_1 y terminando en x_2 . Como el costo del árbol de expansión mínimo $L(AEM)$ es la cota inferior de esta trayectoria, $L(AEM)$ más $c(x_1, x_2)$ es la cota inferior del costo del problema del agente viajero.

En general se desconoce (x_1, x_2) en el circuito óptimo, por tanto se selecciona $\max[c(x_1, s)]$, donde s es el vértice segundo cercano a x_1. Entonces una cota inferior válida de la solución del problema del agente viajero se tiene por medio de:

$$L(AEM) + \max[c(x_1, s)].$$

1.6.3 El PAV y el problema de la ruta más larga

El problema de encontrar la ruta más larga en una red entre un par de vértices específicos no difiere del problema de encontrar una ruta más corta. Y esto es porque los problemas de maximización y minimización pueden convertirse uno en el otro multiplicando la función objetivo por (-1). Y puede, sin embargo, confundirnos porque el problema de la ruta más corta es clasificado como fácil mientras que el problema de la ruta más larga es clasificado como difícil. La razón de esta diferencia es que el problema de la ruta más corta (larga) se hace más fácil, en la medida de que no hay ciclos de duración negativa (positiva). Ya que la duración de los arcos son generalmente positivos. El PAV puede convertirse en un problema de ruta más larga, el cual generalmente contiene ciclos de duración positiva.

Para transformar el PAV en un problema de ruta más larga, primero se transforma la instancia del PAV con c_{ij} como duración de los arcos a la instancia de un problema de circuitos hamiltonianos con duración en los arcos de c_{ij}'. Entonces se reemplaza cada arco de duración c_{ij}' por $c_{ij}'' = M - c_{ij}'$ donde M es moderadamente grande, tan grande, como la suma de las c_{ij}' duraciones de los n valores. Una ruta más larga (sin repetición de vértices) de S a t con respecto a la duración de los arcos c_{ij}'' es un circuito hamiltoniano de S a t. La transformación del problema de la ruta más larga al PAV es ligeramente más complicado.

Como ejemplo, supóngase que se quiere encontrar la ruta más larga del vértice 1 al vértice n, en la digráfica como la que se muestra en la Figura 1.17 (a).

a)

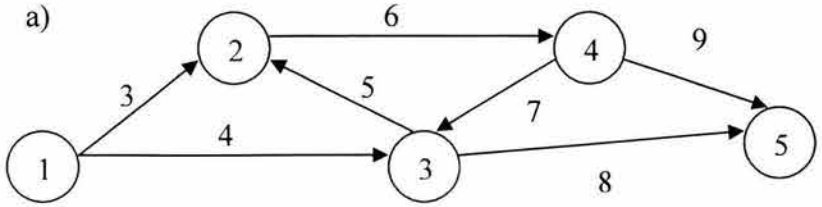
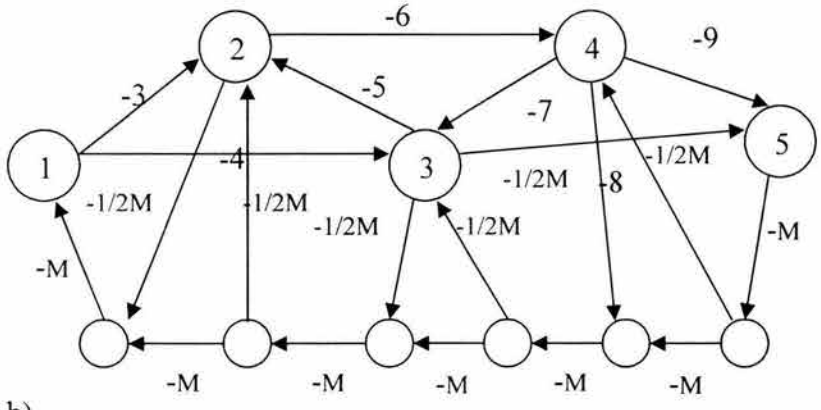


Figura 1.17 (a) Digráfica para el problema de la ruta más larga



b)

Figura 1.17 (b) Transformación de la digráfica al TSP

Primero se multiplica cada duración del arco por (-1), para convertir el problema en un problema de ruta más corta (con ciclos negativos). Entonces se generan $2(n-2)$ nuevos vértices y $4n-7$ nuevos arcos con duraciones de $-M$ y $-\frac{1}{2}M$, donde M es un número muy grande, como se muestra en la figura 1.6 (b). Se afirma que el circuito hamiltoniano, en el nuevo dígrafo tiene la propiedad que la parte del circuito del vértice 1 al vértice n es la ruta más corta (y de un camino más largo en la digráfica original) y la parte del circuito del vértice n al vértice 1 tiene exactamente una duración de $-2(n-2)$.

1.6.4 El PAV y el problema de Asignación Cuadrático (QAP – Quadratic Assignment Problem)

El Problema de Asignación Cuadrática (QAP – Quadratic Assignment Problem) es quizás el más complejo y dificultoso de los problemas de asignación, en donde, relacionar dos asignaciones particulares tiene un costo asociado; tal estructura de costo surge, por ejemplo, cuando el costo de localizar la planta *i* en la localidad *k* y la planta *j* en la localidad *l* es una función de la distancia entre las dos localidades *k* y *l*, y el grado de interacción entre las dos plantas. [www6]

Formalmente, el QAP puede ser definido por tres matrices *nxn*: $D = \{d_{ij}\}$ es la distancia entre la localidad *i* y la localidad *j*; $F = \{f_{hk}\}$ es el flujo entre las plantas *h* y *k*; es decir, la cantidad de interacción (tráfico) existente entre las plantas; $C = \{c_{hi}\}$ es el costo de asignar la facilidad *h* en la localidad *i*. [www6]

Para *n* ciudades con $D = [d_{ij}]$ la matriz de distancias

$$x_{ip} = \begin{cases} 1 & \textit{Si la ruta incluye a la ciudad } i \textit{ en la posicion } p \\ 0 & \textit{De otra forma} \end{cases}$$

Entonces el TSP se puede formular como un problema de asignación cuadrático (*quadratic assignment problem*) de la siguiente forma:

$$\min z \sum_{i,j,p,q=1}^{n} d_{ijpq} x_{ip} x_{jp}$$

s.a.

$$\sum_{p=1}^{n} x_{ip} = 1 \qquad i = 1,\ldots,n$$

$$\sum_{i=1}^{n} x_{ip} = 1 \qquad p = 1,\ldots n$$

$$x_{ip} \in \{0,1\} \qquad i,p = 1,\ldots n$$

Este problema fue enunciado originalmente de una manera ligeramente menos general que la anterior por Koopmans & Beckmann (1957), como un problema de localización de plantas. En este problema

$$d_{ijpq} = c_{ij} t_{pq}$$

Donde t_{pq} es el número de artículos enviados de la planta p a la planta q, y c_{ij} el costo por enviar unidades de la localidad i a la localidad j.

La variable x_{ip} es un indicador de sí o no se asigna la planta p a la localidad i. Es importante hacer notar que la solución del PAV puede ser una permutación cíclica de los enteros $1,\ldots,n$. Entonces x_{ip} puede ser interpretado como un indicador de sí o no, la ciudad i el *p-ésimo* lugar en la permutación (*p-ésima* ciudad visitada). Y aún más el PAV puede ser escrito como un problema de asignación cuadrático en el cual la matriz de distancias están representadas por $C = (c_{ij})$ y la matriz de permutaciones cíclicas por $T = (t_{pq})$. Esto es, $t_{p,p+1} = 1$ para $p = 1,\ldots n-1$, $t_{n1} = 1$ y $t_{pq} = 0$ de otra manera. [Lewler, 1985].

A causa de su diversidad de aplicaciones y a la dificultad intrínseca del problema, el QAP ha sido investigado extensamente por la comunidad científica, clasificándolo como un problema NP – Completo o NP – Hard. [www6].

1.6.5 Problema del cartero chino

Un cartero lleva el correo desde la oficina de correo hasta su destino, es decir lo reparte y cuando regresa a la oficina, el debe por supuesto, cubrir cada una de las calles en esa área al menos una vez. Sujeto a esta condición, el desea escoger su ruta de tal forma que el camine tan poco como sea posible. Este problema es conocido como el Problema del Cartero Chino, desde que fue considerado por el matemático chino Kuan en 1962. En un grafo con peso se define el peso de una ruta $v_0 e_0 e_1 v_1 \ldots e_n v_0$ que será $\sum_{i=1}^{n} w(e_i)$

Claramente se nota que el problema de cartero chino es justamente el encontrar un camino con peso mínimo en un grafo conexo con pesos que no deben ser negativos.

Refiriéndose a este camino como el camino óptimo. Si *G* es Euleriano, entonces cualquier camino Euleriano de *G* es un camino óptimo porque el camino Euleriano es un camino que navega a través de cada vértice exactamente una vez. El Problema del cartero chino se resuelve fácilmente en este caso ya que existe un buen algoritmo para determinar un camino Euleriano en un grafo Euleriano. El algoritmo se debe a Fleury, donde construye un camino Euleriano trazando un rastro, sujeto a una condición, que en cualquier estado, una arista de un subgrafo no-trazado es tomado solamente si no hay alternativa.

1.6.6 Problema del cartero rural

Conocido como Rural Postman
Instancia: El grafo $G=(V,E)$ de longitud $l(e)\in\mathbb{Z}^+$ para cada $e\in E$, subconjunto $E'\subseteq E$, límite $B\in\mathbb{Z}^+$.

Pregunta : ¿Hay un circuito en G que incluye cada arista en E' y que tiene una longitud total no mayor que B?

1.7 Variaciones simples del problema del agente viajero

A continuación se enlistan algunas variaciones del problema del agente viajero que nos permiten encontrar problemas con alta aplicabilidad en la vida real.

1.7.1 El agente viajero MAX o MAXTSP

A diferencia del problema del agente viajero PAV, en este caso se tiene que encontrar un tour en *G* en donde el costo total de las aristas del tour sea máximo. MAX TSP se puede resolver reemplazando cada costo de la arista por uno no negativo, se agrega una constante muy grande por cada arista sin cambiar las soluciones óptimas del problema.

1.7.2 El cuello de botella del PAV

En el problema del PAV, con cuello de botella, el objetivo es minimizar la distancia del recorrido de **mayor duración** que el agente viajero transita, en lugar de minimizar la suma de las distancias de todos los recorridos. [Lawler, 1985]

Como un ejemplo de este problema, se considera una línea de ensamble con estaciones de trabajo arregladas de manera secuencial. Hay *n* actividades para realizar un producto que se mueve a través de la línea de proceso y estas actividades pueden terminar en cualquier orden. El tiempo requerido para realizar la actividad *j* después de la actividad *i*, es:

$$t_{ij} = c_{ij} + p_{ij}$$

Donde c_{ij} es el tiempo de preparación y p_{ij} es el tiempo actual de ejecución de la actividad *j*.

Si el objetivo es la secuencia de actividades y minimizar los tiempos de la línea de ensambles, entonces los criterios del PAV con cuello de botella son apropiados.
Nótese que la optimalidad de una solución al PAV con cuello de botella depende no de las magnitudes de c_{ij} sino únicamente depende de los valores relativos.

Y se plantea de la siguiente manera:

Instancia: Sea un conjunto *C* de m ciudades, la distancia $d(c_i, c_j) \in Z^+$ para cada par de ciudades *ci*, *cj* elementos de *C*, y un entero positivo *B*.

Pregunta: ¿Hay una ruta de *C*, cuya arista más larga no es mayor que *B*, por ejemplo una permutación $< c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(m)} >$ de *C* tal que $d(c_{\pi(i)}, c_{\pi(i+1)}) \leq B$ para $1 \leq i < m$, tal que $d\ c_{\pi(m)}, c_{\pi(1)} \leq B$?

1.7.3 El PAV con ventanas de tiempo (PAVVT)

En este problema, cada vértice *i* tiene una ventana de tiempo asociada $[a_i, b_i]$ uno de los vértices, digamos i_0 , es considerado como un depósito y atravesar un arco $(i, j) \in A$ implica un tiempo para cruzarlo (tiempo de viaje) $t_{i,j} \succ 0$. El PAVVT consiste en encontrar un circuito en *G* empezando en i_0 (el depósito) en el instante a_{i_0} , que atraviese cada vértice exactamente una vez, de tal forma que el circuito debe abandonar cada vértice dentro de su ventana de tiempo y acabe en i_0 antes del instante b_{i_0} . Notar que se permite llegar a un vértice *i* antes de a_i (tiempo de espera), pero en este caso el circuito debe abandonar *i* en el instante a_i . Por simplicidad, si en un vértice *i* es necesario tiempo de servicio, este tiempo está incluido en el tiempo de viaje $a_i t_{j,i}$ $j \neq i$.

1.7.4 El PAV con dependencia horaria (PAVDH)

En los problemas reales de rutas de vehículos, por ejemplo, la distribución dentro de una gran ciudad, además de las ventanas de tiempo de los clientes, el tiempo (que normalmente coincide con el costo) de atravesar algunas calles, como avenidas principales, etc, depende del momento en el que empezamos a atravesarlas.

Por ejemplo, en las horas punta como la entrada/salida del trabajo o del colegio. Si tenemos en cuenta esta idea, los costos de los arcos en algunos problemas de rutas de vehículos deben tener dependencia horaria. En este caso, probablemente casi todos los problemas sencillos que se usan como subrutinas para resolver problemas de rutas (camino más corto, árbol de mínimo peso, acoplamiento, flujo de coste mínimo, etc.) no serían viables.
A pesar de los congestionamientos de tráfico que sufrimos a ciertas horas y en ciertos lugares de las grandes ciudades, los problemas de rutas con dependencia horaria de los costos han sido estudiados muy poco. De hecho el trabajo más reciente sobre este tema, el de Haouari y Dejax (1997), resuelve el problema del camino más corto con ventanas de tiempo y dependencia horaria de los costos en un tiempo pseudo-polinomial.

En los trabajos de Albiach y Soler (2001) y Albiach, et al, (2002) se ha estudiado una generalización del PAVVT que recoge, además de las ventanas de tiempo, la dependencia horaria de los costos. Por esta razón los tiempos de espera están permitidos para minimizar el coste total del viaje. Es decir, está permitido empezar el circuito en el vértice depósito i_0 en el instante $t_{i_0} \succ a_{i_0}$. Por ejemplo, si el instante a_{i_0} está dentro de una hora punta y las restricciones horarias lo permiten, nosotros podemos minimizar el coste total del tour, esperando un breve espacio de tiempo (trabajando en el almacén) en lugar de empezar la ruta en el instante a_{i_0} .

El Problema del Agente Viajero con Dependencia Horaria (PAVDH) se define de la siguiente forma:

Sea $G=(V,A)$ un grafo dirigido, siendo $V=\{v_i\}_{i=0}^{n}$ su conjunto de vértices, donde v_0 es el vértice depósito. Cada vértice $v_i \in V$ tiene asociada una ventana de tiempo $[a_i,b_i]$ verificándose que $a_i,b_i \in Z^+ \cup \{0\}$ y $[a_i,b_i] \subseteq [a_0,b_0]$ $\forall i \in \{1,...,n\}$. Consideramos para cada ventana de tiempo $[a_i,b_i]$ $p_i=b_i-a_i+1$ períodos de tiempo $\{[a_i+k-1,a_i+k]\}_{k=1}^{p_i}$. Por simplicidad denotaremos $T_i^k=[a_i+k-1,a_i+k]$ y con el fin de discretizar el tiempo, identificaremos T_i^k con el instante de tiempo a_i+k-1.

Por otra parte, el tiempo y el costo de atravesar un arco $(v_i,v_j) \in A$ dependen del instante de tiempo $T_i^k (k \in \{1,...,p_i\})$ en el que empezamos a atravesarlo. Denotamos con $T_{i,j}^k \in Z^+$ y $c_{i,j}^k \geq 0$ el tiempo y el costo respectivamente de atravesar el arco (v_i,v_j) empezando en el período T_i^k

El PAVDH consiste en encontrar un circuito hamiltoniano en *G*, que empieza y termina en v_0 dentro de su ventana de tiempo $[a_0,b_0]$ de forma que el circuito abandone cada vértice $v_i \in V$ con $i \succ 0$ dentro de su ventana de tiempo, la suma de los costos sea mínima y con el fin de minimizar el costo total, se permite la espera en cada vértice v_i ,si es alcanzado antes de a_i , siendo esta espera de costo cero. Pero en este caso, el circuito deberá abandonar el vértice en el instante primero a_i .

Como en el PAVVT, por simplicidad asumimos que el tiempo de atravesar un arco (v_i,v_j) con $j \succ 0$ incluye el tiempo de servicio en v_j . En el caso particular de un PAVDH en el que $T_{i,j}^k = T_{i,j}^s = c_{i,j}^k = c_{i,j}^s \forall k,s \in \{1,...,p_i\}$ y $\forall (v_i,v_j) \in A$, tenemos un PAVVT con la función objetivo igual al tiempo total del circuito. Así, el PAVDH es un problema NP-duro.

1.7.5 Múltiples agentes viajeros

El problema del viajero clásico considera que un solo viajero debe realizar la visita a todas las ciudades. En el problema de viajeros múltiples se disponen de m viajeros. Se puede suponer que existe un costo adicional fijo por tener activo un viajero. El problema es determinar cuántos viajeros usar y cuales son las rutas que recorre para que el costo total sea mínimo.

El problema de viajeros múltiples tiene diferentes variaciones. Estas se clasifican de la siguiente manera:

- Todos los viajeros tienen que partir de una "ciudad base"
- Al menos *r* viajeros tienen que ser activos
- En un caso más general, los viajeros pueden partir de diferentes ciudades

En [Jianyu, 1985], se hace un análisis de cada uno de estos casos.

Se verá a continuación el planteamiento matemático de este problema.
Se asume que m agentes viajeros se localizan en el nodo 1 de G. Cada agente viajero visita un subconjunto X_i de nodos de G, comenzando por el nodo 1, visitando cada ciudad exactamente una vez y regresando al nodo 1. Se esta interesado en encontrar una partición $X_1, X_2, ..., X_m$ de $V-\{1\}$ y una ruta de cada m agente viajero tal que: $(i)|X_i| \geq 1$ para cada i, $(ii)\bigcup_{i=1}^{m} X_i = V-\{1\}$, $(iii) X_i \cap X_j = \varnothing$ para $i \neq j$ y (iv) minimizar la distancia total realizada por todos los agentes viajeros. Si G es no dirigida, los m agentes viajeros del problema del agente viajero se pueden formular como un problema del agente viajero sobre $G'=(V',E')$ con $m-1$ nodos adicionales, donde $V'=V\cup\{n+2,n+3,...,n+m\}$, $E'=E\cup\{(i,n+k):2\leq k\leq m, 2\leq i\leq n\}$. Sea c'_{ij} el costo de las aristas en G', donde $c'_{ij}=c_{ij}$ si $(i,j)\in E$ y $c'_{i,n+k}=c_{i1}$ para $2\leq k\leq m, 2\leq i\leq n$. Se puede recobrar una solución óptima de m-múltiples agentes viajeros sobre una solución óptima del PAV sobre G'. El caso cuando G es una gráfica dirigida se puede manejar de manera similar con algunas modificaciones simples como la anterior.

1.8 Complejidad computacional

Prácticamente todas las áreas de las ciencias computacionales tratan, en mayor o menor grado, con complejidad computacional. El tema es muy común entre expertos del área, sobre todo entre aquéllos relacionados con **NP-completos** y entre quienes buscan algoritmos eficientes para diversos problemas de aplicación. La importancia de la complejidad computacional estriba en que se ha convertido en una forma de clasificar buenos y malos algoritmos y de clasificar problemas computacionales como fáciles y difíciles.

El problema del agente viajero fue uno de los primeros donde se aplicó la teoría de la *NP-Completez* a principios de los 70´s. A partir de entonces se ha usado como el ejemplo prototipo de los problemas combinatorios *NP-difíciles*. Además, el PAV ha dado pie al desarrollo de nuevos algoritmos. Por ejemplo, el método de relajación lagrangeana se desarrolló a partir del trabajo de Held y Karp, para resolver el problema del agente viajero.

Una vez que se sabe que el problema es NP-difícil, y por lo tanto es improbable que exista un algoritmo polinomial que encuentre la solución óptima, se buscan algoritmos aproximados eficientes. Es decir, algoritmos que encuentran una solución cercana a la óptima en un tiempo corto. Hasta la fecha, los mejores algoritmos de este tipo son aquéllos que se basan en una técnica conocida como optimización local, en la que una solución se mejora continuamente al realizar cambios locales.

1.8.1 Teoría de NP-Completez

Existen muchos problemas que no se pueden resolver con las técnicas disponibles de manera exacta y eficiente. Algunos de estos problemas podrían ser resueltos con algoritmos eficientes que aún no se descubren. Sin embargo, es muy probable que muchos de ellos no puedan ser resueltos eficientemente. Entonces, es de gran utilidad identificar este tipo de problemas con el propósito de no invertir tiempo en buscar algoritmos que no existen. La teoría de la NP-Completez proporciona técnicas para identificar este tipo de problemas.

A continuación se definen algunos conceptos que se utilizarán más adelante.

Un *problema* se especifica con la descripción general de sus parámetros y las propiedades que debe tener la solución. Como ejemplo considérese el problema del agente viajero. Los parámetros de este problema son las ciudades $1,2,\ldots,n$ y para cada par de ciudades i,j la distancia $c_{i,j}$ entre ellas. La solución es una permutación $(\pi_1,\pi_2,\ldots,\pi_n)$ de ciudades que minimicen $\sum_{i=1}^{n-1} c_{\pi_i\pi_{i+1}} + c_{\pi_m\pi_1}$.

Un *ejemplo* de un problema se obtiene al especificar los valores de todos los parámetros del problema. Así, un *ejemplo* para el problema del agente viajero está dada por $\{1,2,3,4\}, c_{12}=10, c_{13}=5, c_{14}=9, c_{23}=6, c_{24}=9$ y $c_{34}=3$.

La permutación $(1,2,3,4)$ es una solución de este *ejemplo* con costo total de 23 unidades.

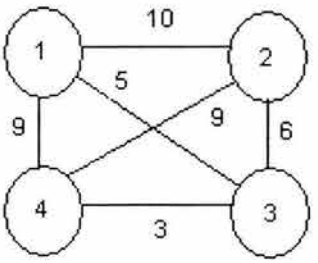


Figura 1.18 E*jemplo* del problema del agente viajero

Un *algoritmo* es un conjunto de instrucciones o reglas bien definidas que se usan para obtener un resultado específico a partir de unos datos de entrada específicos en un número finito de pasos.

Se está interesado en encontrar el algoritmo más "eficiente" que resuelva un problema. La noción de eficiencia involucra todos los recursos de cómputo que se necesitan para ejecutar un algoritmo. Sin embargo, el algoritmo "más eficiente" generalmente significa el más rápido. El tiempo requerido por un algoritmo depende del tamaño del *ejemplo* del problema. El tamaño refleja la cantidad de datos de entrada necesarios para describir el *ejemplo*.

Generalmente esta medida se hace de manera informal. Por ejemplo, para el problema del agente viajero el número de ciudades se toma como el tamaño del *ejemplo*, aunque existan otros datos de entrada como la distancia entre cada par de ciudades.

La *función de complejidad de tiempo* proporciona el mayor tiempo requerido por un algoritmo para resolver un *ejemplo* de un problema con un determinado tamaño. Los algoritmos tienen una gran variedad de funciones de complejidad de tiempo y determinar cuáles algoritmos son suficientemente eficientes y cuáles son muy ineficientes depende de cada situación. Sin embargo, éstos se han dividido en algoritmos de tiempo polinomial y algoritmos de tiempo exponencial. En la comunidad científica estos algoritmos los han dividido como buenos y malos algoritmos.

1.8.2 Buenos y malos algoritmos

Los algoritmos han sido divididos como buenos o malos algoritmos. La comunidad computacional acepta que un buen algoritmo es aquél para el cual existe un algoritmo polinomial deterministico que lo resuelva. También se acepta que un mal algoritmo es aquel para el cual dicho algoritmo simplemente no existe. Un problema se dice intratable, si es muy difícil que un algoritmo de tiempo no polinomial lo resuelva.

Se dice que una función $f(n)$ es de orden $g, O(g(n))$, si existe una constante c tal que $|f(n)| \prec c|g(n)|$ para $n \geq 0$. Un *algoritmo de tiempo polinomial* se define como aquél cuya función de complejidad de tiempo sea $O(p(n))$ para alguna función polinomial p y donde n es el tamaño del *ejemplo* del problema. Los problemas de la clase P son aquéllos para los cuales existe un algoritmo polinomial que los resuelva. Cualquier algoritmo cuya función de complejidad de tiempo no pueda ser acotada de esta manera se conoce como *algoritmo exponencial.* Cabe hacer notar que esta definición incluye ciertas funciones no polinomiales como $n^{\log n}$, que no se consideran como funciones exponenciales.

No obstante, esta clasificación de algoritmos en buenos y malos resulta a veces engañosa, ya que se podría pensar que los algoritmos exponenciales no son de utilidad práctica y que habrá que utilizar solamente algoritmos polinomiales.

Por ejemplo; un algoritmo de complejidad n^{80} tomará para resolver instancias de tamaño 3 tiempos astronómicos, mientras que un algoritmo exponencial correrá más rápidamente para toda instancia razonable.

Sin embargo, *la experiencia ha demostrado que para la mayoría de los problemas una vez que un algoritmo acotado en tiempo polinomial es descubierto, el grado del polinomio rápidamente sufre una serie de decrementos tan pronto como varios investigadores mejoran la idea.* Generalmente, la razón final de crecimiento es $O(n^3)$ o mejor. Por ejemplo, se tiene el caso de los *métodos simplex y Branch & Bound*, los cuales son muy eficientes para muchos problemas prácticos. Desafortunadamente ejemplos como estos dos son raros, de modo que es preferible seguir empleando como regla de clasificación en buenos y malos algoritmos, dependiendo de si son o no polinomiales; todo esto con la prudencia necesaria.

Es obvio que, cuando el tamaño de la entrada crece, cualquier algoritmo polinomial eventualmente llegará a ser más eficiente que cualquier algoritmo exponencial.

Una característica positiva de los algoritmos polinomiales es que toman más ventaja de los avances de la tecnología. Por ejemplo, cada vez que una mejora tecnológica incrementa la velocidad de las computadoras 10 veces, el tamaño de la instancia más grande que un algoritmo polinomial soluciona en una hora, por ejemplo, será multiplicado por una constante entre 1 y 10. En contraste, un algoritmo exponencial experimentará únicamente un incremento pequeño que se sumará al tamaño de la instancia que éste puede resolver. (Ver tabla 1.3).

Función	Tamaño de la Instancia solucionada en un día	Tamaño de la Instancia solucionada en un día en una computadora 10 veces más rápida
n	10^{12}	10^{13}
n log n	0.948×10^{11}	0.87×10^{12}
n^2	10^6	3.16×10^6
n^3	10^4	2.15×10^4
$10^8 n^4$	10	18
2^n	40	43
10^n	12	13
$n^{\log n}$	79	95
n!	14	15

Tabla 1.3 Algoritmos de tiempo polinomial toman más ventaja de los avances de la tecnología.

Finalmente se puede decir que los algoritmos polinomiales tienen la propiedad de cerradura: pueden ser combinados para resolver casos especiales del mismo problema; un algoritmo polinomial puede llamar otro algoritmo polinomial como una subrutina y el algoritmo resultante continuará siendo polinomial.

1.8.3 Problemas fáciles y difíciles

Sin entrar en detalles técnicos, decimos que un problema es "fácil" de resolver cuando es posible encontrar un algoritmo (método de solución) cuyo tiempo de ejecución en una computadora crece de forma "razonable" o moderada (o polinomial) con el tamaño del problema. Por el contrario, si no existe tal algoritmo decimos que es "difícil" de resolver.

Esto no implica que el problema no pueda resolverse, sino que cada algoritmo existente para la solución del problema tiene un tiempo de ejecución que crece explosivamente (o exponencialmente) con el tamaño del problema, el tiempo requerido para la solución aumenta de forma exponencial, lo cual limita el tamaño de problemas que pueden resolverse en las computadoras modernas. Técnicamente, determinar si un problema es fácil o difícil se denomina establecer la ***complejidad computacional*** del problema, y esto es todo un arte, especialmente para demostrar que un problema es de los difíciles. [González, 1999]
Como se comentaba en párrafos anteriores, el problema del agente viajero, como parte de un problema combinatorio, una forma de obtener la respuesta a este problema es mediante una enumeración exhaustiva. Es decir, formamos todas las posibles combinaciones de "tours", en este caso (n-1)!, donde n!=n(n-1)(n-2)...(2)(1)) y calculamos la distancia total para cada "tour", eligiendo aquel que tenga la mínima distancia total. En este caso el problema ha quedado totalmente resuelto porque estamos exhibiendo todos los tours posibles. El tiempo de ejecución de este algoritmo es grosso modo $f(n) = (n)!$. [González, 1999]

Hay que notar que la función factorial $f(n)=(n)!$, es una función que crece exponencialmente a medida que crece el valor de n. Claro, esto no prueba que el PAV es difícil, ya que muy bien pudiera existir otro algoritmo que lo resolviera, cuyo tiempo de ejecución fuera polinomial. En este caso, sin embargo, **ya se ha demostrado que tal algoritmo polinomial no existe y que el PAV pertenece a esa clase de problemas difíciles.**

En el estudio de la existencia de algoritmos que permitan encontrar la solución buscada en un tiempo polinomial y la construcción de ellos cuando es posible, es muy importante el conocimiento de las propiedades y estructura matemática del problema. En particular, la teoría de gráficas permite, en muchos casos, el estudio de esta estructura y al aprovechar sus propiedades es posible construir los algoritmos buscados.

La tabla 1.4 tomada de [Garey, 1979]], ilustra las diferencias de crecimiento de diferentes funciones de tiempo (columnas). Las cifras que se muestran son de tiempo de procesamiento en computadora que procesa 1 millón de operaciones de punto flotante por segundos. Nótese el crecimiento explosivo de las funciones exponenciales. (Últimas columnas).

Tamaño n	$f(n)=n$	$f(n)=n^2$	$f(n)=n^3$	$f(n)=n^5$	$f(n)=2^n$	$f(n)=3^n$
10	.00001 seg	.0001 seg	.001 seg	.1 seg	.001 seg	.059 seg
20	.00002 seg	.0004 seg	.008 seg	3.2 seg	1.0 seg	58 minutos
30	.00003 seg	.0009 seg	.027 seg	24.3 seg	17.9 minutos	6.5 años
40	.00004 seg	.0016 seg	.064 seg	1.7 minutos	12.7 dias	3857 siglos
50	.00005 seg	.0025 seg	.125 seg	5.2 minutos	35.7 años	2×10^8 años
60	.00006 seg	.0036 seg	.216 seg	13 minutos	366 siglos	1.3×10^{13} siglos

Tabla 1.3 Comparación de varias funciones polinomiales y exponenciales

1.8.4 Clase NP, NP-completa y CoNp.

La teoría de la NP-Completez proporciona técnicas para demostrar que un problema es tan "difícil" como un gran número de problemas que son conocidos como "difíciles". La principal técnica que se usa para demostrar que dos problemas están relacionados es "reducir" un problema en otro, al transformar cada *ejemplo* de un problema en un *ejemplo* equivalente de otro problema. Para ello se considerarán problemas de decisión, es decir, problemas en donde la respuesta es *sí* o *no*. El objetivo es restringirse a estos problemas es hacer la teoría más simple. Muchos problemas combinatorios se pueden transformar en problemas de decisión. Por ejemplo, el problema de decisión para el problema del agente viajero es:

Pregunta: ¿Hay una ruta de C que tenga una longitud B o menor, por ejemplo una permutación $<c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(k)}, c_{\pi(k+1)}, \ldots, c_{\pi(m)}>$ de C tal que :

$$\left(\sum_{i=1}^{m-1} d\left(c_{\pi(i)}, c_{\pi(i+1)}\right) + d\left(c_{\pi(m)}, c_{\pi(1)}\right) \right) \leq B ?$$

Se introducirá el concepto de reducción polinomial. Supóngase que se tiene un problema π_1 que puede ser resuelto por por un algoritmo A. Si cada instancia de un problema π_2 se puede transformar en un *ejemplo* de π_1 en un tiempo polinomial, entonces claramente puede ser usado este hecho y el algoritmo A para resolver π_2. Si el número de instancias del problema π_1 es mayor o igual que el número de instancias transformadas del problema π_2, éstas se pueden ver como casos particulares de las instancias de π_1. Entonces, es razonable afirmar que el problema π_1 es tan difícil (o posiblemente más difícil) que π_2.

Los problemas de decisión pueden tener diferentes grados de dificultad. Pero si se tuviera una solución candidata, sería fácil verificar si con ésta se demuestra que la respuesta al problema de decisión es *sí*. Para los problemas de decisión que están en *NP* no se requiere que cada *ejemplo* del problema pueda ser resuelto en un tiempo polinomial por algún algoritmo. Sólo se requiere que para las instancias del problema en la cual la respuesta al problema de decisión es *sí*, exista una solución con la que se pueda verificar la respuesta en tiempo polinomial. Específicamente, la *clase NP* incluye aquellos problemas de decisión que pueden ser resueltos en tiempo polinomial, si se "adivina" la solución con la que se puede demostrar que el problema de decisión es *sí*. Las letras *NP* significan *polinomial no determinístico*. Es decir, los problemas en *NP* se resuelven en tiempo polinomial no determinístico, en el sentido de que se pueden generar soluciones candidatas con una alta probabilidad de adivinar alguna que sirva para demostrar que la respuesta al problema de decisión es *sí*. [Aguirre, 1996]

El complemento de un problema de decisión se obtiene al intercambiar los papeles de las respuestas *sí* o *no*. Por ejemplo, el complemento del problema de decisión para el problema del agente viajero es el siguiente:

¿Hay una ruta de C que tenga una longitud B o menor, por ejemplo una permutación $<c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(k)}, c_{\pi(k+1)}, \ldots, c_{\pi(m)}>$ de C tal que :

$$\left(\sum_{i=1}^{m-1} d\left(c_{\pi(i)}, c_{\pi(i+1)}\right) + d\left(c_{\pi(m)}, c_{\pi(1)}\right) \right) \leq B ?$$

Obsérvese que las instancias para las cuales la respuesta al problema de decisión es *sí* tienen como respuesta *no*, cuando se considera el complemento del problema de decisión.

El complemento de los problemas de la clase *NP* pertenece a la clase *CoNP*. ¿Cómo se pueden resolver problemas que pertenecen a esta clase? Es decir, ¿cómo se puede probar que la respuesta al problema de decisión es no? La única manera es enunciar todas las posibles soluciones y verificar que con ninguna de éstas se puede demostrar que la respuesta al problema de decisión es *sí*. Dado que esta enumeración se realiza en tiempo

exponencial no se puede verificar que la respuesta es no en tiempo polinomial, es decir, existen problemas en *CoNP* que no están en *NP*.

Si un problema π es tal que cualquier problema en NP se puede reducir polinomialmente a π, se dice que π es *NP-difícil*. Si además el problema π pertenece a la clase NP, entonces π es *NP-completo*. Por lo tanto, los problemas de clase *NP-completa* son los más difíciles de la clase *NP*. Los problemas *NP-completos* son importantes, ya que si se encuentra un algoritmo polinomial para resolver alguno de ellos se tendrá un algoritmo polinomial para todos los problemas en *NP*; es decir, se habrá mostrado que *P=NP*.

Puede parecer sorprendente que exista un problema par el cual cualquier problema en NP se puede reducir a él. Sin embargo, Cook demostró en 1971 que el problema de satisfactibilidad (SAT) es NP-completo, es decir, demostró que la clase NP-completa no es vacía. El problema de satisfactibilidad es el siguiente. Sea S una expresión lógica que está formada por el producto de varias sumas. Por ejemplo, $S = (x_1 + x_2 + \overline{x}_3) \bullet (\overline{x}_1 + x_2 + x_3) \bullet (\overline{x}_1 + \overline{x}_2 + \overline{x}_3)$, donde las sumas y las multiplicaciones corresponden a las operaciones lógicas *y* y *o* respectivamente y donde cada variable vale 0 (falso) ó 1 (verdadero). El complemento de la variable x_i se denota por $\overline{x}_i$. Se dice que una expresión lógica se satisface, si existe una asignación de ceros y unos de las variables tal que el valor de la expresión sea 1. El problema SAT consiste en determinar si una expresión lógica se satisface. Por ejemplo, la expresión S sí se satisface, lo cual puede verificarse con la siguiente asignación: $x_1 = 1, x_2 = 1$ y $x_3 = 0$.

En 1972 Karp demostró que existen otros problemas que pertenecen a la clase *NP-completa*. Una vez que se ha encontrado un problema *NP-completo* es más fácil demostrar que otros problemas también pertenecen a la clase usando la siguiente observación: un problema π_1 es *NP-completo*. En la tesis *Propiedades y algoritmos para el problema del agente viajero* de Rosalía Aguirre Hernández, del año 1996 en las páginas de la 16 a la 24 se realizan las transformaciones siguientes:

SAT → *clique* → recubrimiento de vértices → circuito hamiltoniano → problema del agente viajero.

para demostrar que el problema del agente viajero es *NP-completo*. En este trabajo se realizará la reducción del problema del circuito hamiltoniano al problema del agente viajero.

1.8.5 El Problema del agente viajero como un problema NP-Completo

Decimos que para que un problema sea NP-Completo tiene que cumplir dos condiciones:

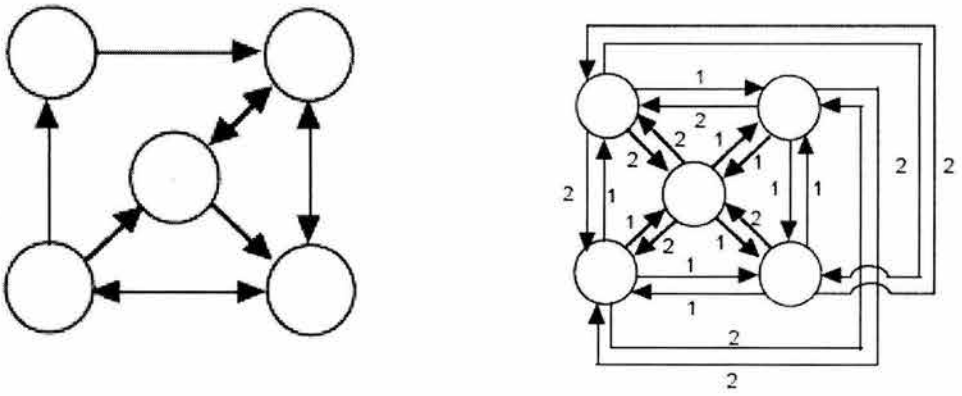
a) Tiene que ser un problema NP, es decir, mostrar que el PAV ($\prod$) pertenece a NP.
b) Se tiene que encontrar un problema $\prod'$ que se conoce es NP-Completo tal que $\prod'$ sea polinómicamente reducible al PAV ($\prod$). [www5]

Entonces, para demostrar que el PAV es NP-Completo se tiene que probar que el PAV pertenece a NP y que el problema del Circuito Hamiltoniano ($\prod'$) que se conoce es un problema *NP-Completo*. Solamente hay que realizar la reducción del problema del circuito

hamiltoniano al problema del agente viajero, para demostrar que el problema del agente viajero es *NP-completo*.

Para hacerlo, construimos un grafo nuevo G'. Donde G' tiene los mismos vértices que el grafo G. Para G', cada arista *(i,j)* tiene un peso de 1 si *(i,j)* $\in G$, y un peso de 2 en cualquier otro caso. O sea, transformamos un problema del ciclo hamiltoniano en un problema del agente viajero.

Ya que todas las rutas contienen *n* aristas, la existencia de una ruta de costo *n* implica que cada una de las aristas incluidas tengan un costo de 1, esto es, cada una de las aristas incluida en la ruta aparece en la instancia HC. Así, una ruta de costo *n* implica una solución para la instancia HC. A la inversa, si hay una solución HC, entonces cada una de las aristas que aparecen en la solución tienen un costo de 1 en la instancia PAV, y se encuentra así, una solución para el PAV de costo *n*. En la Figura 1.119.b se presenta la instancia PAV correspondiendo a la instancia HC de la figura 1.11.a.



a) Una instancia del problema del circuito Hamiltoniano

b) Una instancia equivalente del PAV

Figura 1.19 Reducción polinomial del problema del circuito hamiltoniano al problema del agente viajero (PAV)

Es sencillo verificar que G tiene un ciclo hamiltoniano, si y solo si G' tiene un tour de peso total $|V|$ (o lo que es lo mismo n). Por lo tanto, el problema del ciclo hamiltoniano es polinomicamente reducible al problema del agente viajero, por lo cual, se demuestra que el Problema del Agente Viajero es NP-Completo.

1.8.6 El Problema del agente viajero desde la complejidad computacional

Como he comentado en párrafos anteriores que, determinar si un problema es fácil de resolver cuando es posible encontrar un algoritmo cuyo tiempo de ejecución en una computadora crece de forma razonable o polinomial con el tamaño del problema. Y si no existe tal algoritmo decimos que es difícil de resolver, y en este caso el Problema del Agente Viajero es un problema difícil porque hasta el momento no existe un algoritmo que lo resuelva el problema en forma eficiente y en un tiempo razonable (polinomial).

Al saber que un problema es NP-completo se pueden tomar varios caminos. Si el problema es pequeño entonces se puede resolver eficientemente con algún algoritmo exacto. Pero si el problema no es pequeño la búsqueda de un algoritmo eficiente y exacto no es prioritario. Es más apropiado concentrarse en metas menos ambiciosas. Por ejemplo, buscar algoritmos eficientes que resuelvan casos particulares del problema general. Buscar algoritmos que aunque no exista garantía de ser eficientes lo sean en la mayoría de los casos. Relajar el problema y buscar algoritmos "heurísticos" eficientes los cuales aunque no garanticen obtener la solución óptima obtienen una cercana a ésta.
Finalmente se puede decir que la solución del problema del agente viajero nos permite una doble interpretación, una permutación o un circuito hamiltoniano.

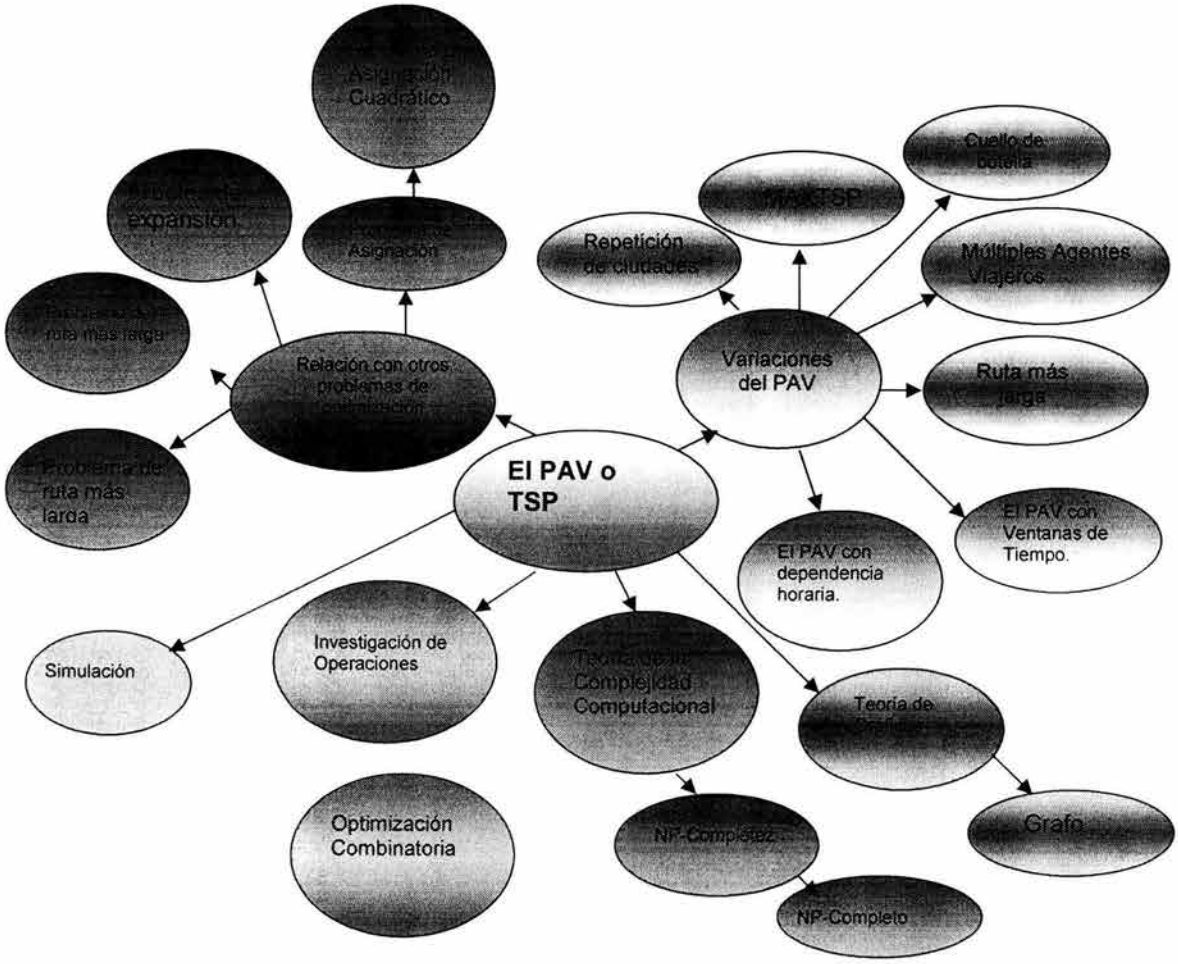


Figura 1.20 Marco Conceptual del PAV

Capítulo 2. Técnicas exactas para resolver el problema del agente viajero

En este capítulo describiré dos de las técnicas exactas que se han desarrollado para resolver problemas de optimización combinatoria. Como lo son el método de Ramificación y Acotamiento (R-A) o Branch and Bound y la programación dinámica. Así, primero doy una descripción de la manera en que trabajan los algoritmos exactos, acompañados de algunos ejemplos. Cabe mencionar que el método de ramificación y acotamiento más usual y clásico es el método de R-A basado en la matriz reducida, aunque se tienen los combinados con árbol de expansión mínima y problema de asignación, entre otros.

Introducción

Como se podrá ver el problema del agente viajero es uno de los problemas de optimización combinatoria para los cuales no se conoce un algoritmo eficiente de tiempo polinomial. Todos los algoritmos conocidos requieren de un tiempo de computadora exponencial en el número *n* de ciudades. Es decir, formamos todas las posibles combinaciones de "tours", en este caso $(n-1)!$, donde $n! = n(n-1)(n-2)...(2)(1)$ y calculamos la distancia total para cada "tour", eligiendo aquel que tenga la mínima distancia total. En este caso el problema ha quedado totalmente resuelto porque estamos exhibiendo todos los tours posibles. El tiempo de ejecución de este algoritmo es grosso modo $f(n) = (n)!$. Y esta función programada en una computadora nos da un costo exponencial.

Para salvar esta dificultad computacional hay dos formas de resolverla:

1. Usando técnicas refinadas tales como ramificación y acotamiento o programación dinámica, que reducen drásticamente el efecto que se da en enumeración exhaustiva. Tales técnicas refinadas enumerativas son buenas para encontrar una solución óptima, pero en el peor de los casos si se tiene un problema muy grande pueden requerir un número exponencial de cálculos que se hacen prohibitivamente grandes.

2. Empleando métodos de solución aproximados pero rápidos (en tiempo polinomial), que no producen una solución óptima, pero si soluciones subóptimas que estén aceptablemente cercanas a la óptima.

Ambos métodos son útiles, sin embargo para este capítulo se discuten las técnicas que se señalan en el punto número uno.

2.1 Funcionamiento de un algoritmo exacto

En el caso más general, se asume que la solución a un problema consiste de un vector $(a_1, a_2, ...)$ de longitud finita pero indeterminada, satisfaciendo ciertas restricciones. Cada a_i es un miembro de un conjunto A_i linealmente ordenado. Así la búsqueda exhaustiva debe considerar los elementos $A_1 \, x \, A_2 \, x \, ... \, x \, A_i$, para $i = 0, 1, 2, ...$, como solución potencial. Primero se inicia con el vector nulo $()$ como la solución parcial, y las restricciones indican cual de los miembros de A_1 son candidatos para a_1; llamado este subconjunto S_1. Se escoge el menor elemento de S_1 como a_1, y se tiene ahora la solución parcial (a_1). En general, las diversas condiciones que describen las soluciones nos indican cual subconjunto

S_k de A_k constituye candidatos para la extensión de la solución parcial $(a_1, a_2, ..., a_{k-1})$ para $(a_1, a_2, ..., a_{k-1}, a_k)$. Si la solución parcial $(a_1, a_2, ..., a_{k-1})$ no admite posibilidades para a_k, entonces $S_k = \varnothing$, y así se regresa hasta el nodo padre y se hace una nueva selección para a_{k-2}, y así sucesivamente. [Castañeda, 2000]

El subconjunto de $A_1 \, x \, A_2 \, x \, ... \, x \, A_i$ para $i = 0, 1, 2, ...,$ es representado como un árbol de búsqueda como se puede ver en la figura 2.1. La raíz del árbol (el nivel 0) es el vector nulo. Sus hijos son la selección para a_1; y en general, los nodos como el k-ésimo nivel son las selecciones para a_k, dando como resultado las selecciones hechas por $a_1, a_2, ..., a_{k-1}$ como indicadores por los ancestros de estos nodos. En el árbol presentado en la figura 2.1 recorrer los nodos hacia atrás como se indica con las líneas punteadas. [Castañeda, 2000]

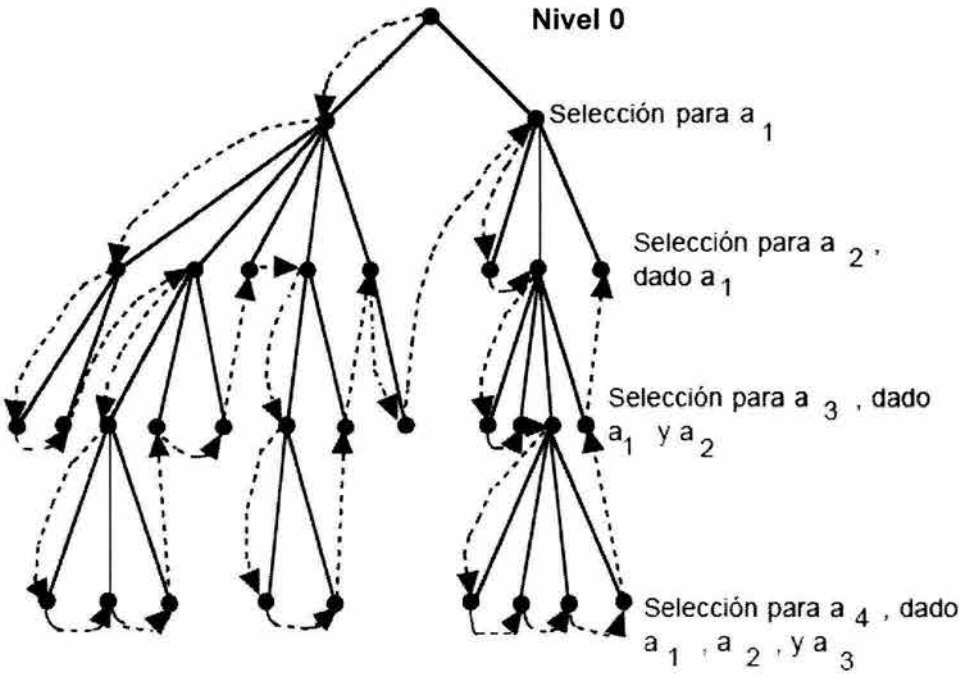


Figura 2.1 Árbol de búsqueda de las soluciones parciales

Cuando se pregunta si un problema tiene una solución $(a_1, a_2, ...)$ en realidad se está preguntando si cualquier nodo en el árbol es una solución. Si se pregunta por todas las soluciones, se desean tener todos esos nodos.

2.2 Método de Ramificación y Acotamiento (Branch and Bound)

El método de Ramificación y Acotamiento (R-A) surgido hace tres décadas, ha resultado ser una de las mejores herramientas prácticas para la solución de problemas de optimización discreta. Su atractivo radica en la habilidad de eliminar implícitamente grupos grandes de soluciones potenciales sin evaluarlos explícitamente. A semejanza de la programación dinámica, la técnica de ramificación y acotamiento es una estrategia, y como tal se debe

combinar con la estructura del problema específico que se desea resolver, para así formar un algoritmo de solución adecuado.

Hay muchos problemas de optimización discreta para los cuales los métodos "directos" o no existen o son ineficaces. Los problemas pueden ser tales que las restricciones o función(es) objetivo(s) son no convexas, o que todos o algunos de los valores están restringidos a valores discretos. La técnica de ramificación y acotamiento nos posibilita resolver problemas "difíciles" usando los métodos existentes para la solución de problemas "fáciles".

La estrategia de aplicación de método de R-A se basa en la premisa de que el problema que se va a resolver tenga los siguientes atributos:

a) **Naturaleza Combinatoria**: Un problema combinatorio tiene como mínimo las siguientes propiedades:

a.1) Se da un conjunto finito de objetos
a.2) Cada objeto puede tomar un cierto rango de atributos.
a.3) Se desarrolla una solución al problema fijando los valores de atributos para todos los objetos
a.4) Solo se permiten ciertas combinaciones de los valores de atributos.

b) **Ramificabilidad**: Es una propiedad del problema que implica:

b.1) Construir un conjunto finito y contable que contenga todas las soluciones del problema (esto se sigue de 1);
b.2) Particionar recursivamente un conjunto no vacío de soluciones en subconjuntos disjuntos.

c) **Racionalidad**: Un problema racional es tal que:

c.1) Cada solución tiene un único valor calculado de los valores de sus atributos.
c.2) La "mejor" solución es aquella con mayor (o menor) valor.

d) **Acotabilidad**: Una estimación del valor de la mejor solución contenida en cualquier conjunto de soluciones se puede obtener tal que:

d.1) El valor actual de la mejor solución en el conjunto es inferior o igual a la estimación (así la estimación es una cota superior);
d.2) Se hace un mínimo esfuerzo para obtener dicha estimación;
d.3) La estimación es razonablemente cercana al valor actual.

El concepto de R-A explota estas propiedades para poder implícita y explícitamente construir un árbol que describa todas las operaciones realizadas en el problema, y efectuar una búsqueda para encontrar la mejor solución.

2.2.1 El árbol de Búsqueda

El procedimiento de ramificación y acotamiento (R-A) proporciona una metodología de búsqueda de la solución óptima en un problema de optimización discreta. En el método de R-A, el conjunto de soluciones factibles se parte en subconjuntos más simples (Esto es lo que se debería hacer en la práctica, sí uno está buscando por ejemplo una aguja en un pajar. El pajar es grande y es imposible buscar en todo simultáneamente, así que se puede dividir visualmente en lado derecho e izquierdo y seleccionar uno de ellos para buscar la aguja, manteniendo el otro lado en espera para buscar después si es necesario). A

continuación un subconjunto prometedor se selecciona y se hace un esfuerzo para encontrar la mejor solución factible y se almacena esta información, en algunos casos podría darse por terminado el método. Se parte nuevamente el subconjunto en dos o más subconjuntos más simples (bajo la operación denominada ramificación) y se repite el mismo proceso. [Flores, 2002]

En [Aguirre, 1996] se menciona que la técnica de (R-A) es un procedimiento de *enumeración parcial* ya que examina un pequeño número de soluciones, eliminando algunas soluciones no prometedoras antes de ser avaluadas.

El proceso de ramificación del problema de optimización discreta puede representarse por medio de un árbol de búsqueda. Cada nodo del árbol está asociado a un subconjunto de T, conjunto de todas las soluciones originales del problema. Sea T_i un subconjunto de T, esto es, una colección de soluciones del problema. La Ramificación es el proceso de partir un subconjunto T_i en m subconjuntos disjuntos $v_1, v_2, ..., v_m$ donde

$$v_1 \bigcup v_2 \bigcup v_3 ... \bigcup v_m = T$$
$$v_i \bigcap v_j = \varnothing, \quad i \neq j$$

O bien, la unión de ellos es T_i y la intersección entre ellos es vacía.

El proceso de ramificación se puede visualizar como la creación o desarrollo ordenado de un árbol donde el nodo inicial representa a T, y los nodos restantes representan subconjuntos T_i de T. A cada nodo N se le asocia un subconjunto T.

En la Figura 2.2 se ilustra un árbol que exhibe estos conceptos para el caso $m = 2$. El nodo N =1 corresponde a T, el conjunto de todas las soluciones. Los nodos 2 y 3 están asociados con los conjuntos ajenos T_1 y T_2 de T. Note que no hay ramificaciones desde los nodos 3, 4 y 5.

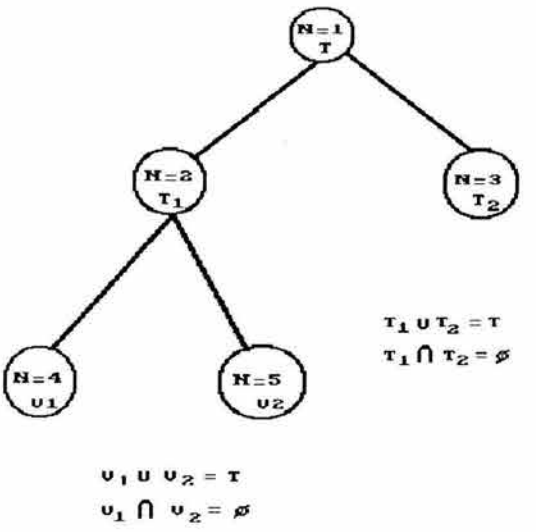


Figura 2.2 Árbol de búsqueda para m=2

La relación, entre subconjuntos de T y la ramificación para la creación del árbol, se formaliza como sigue:

$T_i(N)$ es el subconjunto T_i de T representado por el nodo N.

Un nodo intermedio es un nodo N en el cual no ha habido ramificación.

Un nodo final es un nodo intermedio N para el cual $T_i(N)$ consiste de una solución única. $L(N)$ es una cota inferior de la función objetivo $Z(x)$ en el conjunto de las soluciones asociadas con el nodo N, esto es, $L(N) \leq Z(x)$ para toda $x \in T_i$.

Una forma conceptual y al mismo tiempo general de describir el método de Ramificación y Acotamiento es con el siguiente algoritmo:

2.2.2 El algoritmo básico de Ramificación y Acotamiento

Propósito: Encontrar la solución óptima de un problema de optimización discreta

Descripción

Paso 0: Se inicia con el conjunto de todas las soluciones factibles del problema en cuestión. Se forma el primer nodo del árbol.

Paso 1: Se procede a ramificar los nodos hacia nodos nuevos, utilizando alguna regla de ramificación.

Paso 2: Se determinan cotas inferiores para los nuevos nodos. Para cada nuevo nodo N se obtiene una cota inferior $L(N)$ sobre el valor de la función objetivo para las soluciones factibles del nodo.

Paso 3: Se selecciona un nodo intermedio desde el cual se ramifica. Cada nuevo nodo se excluye o se considera examinado si se encuentra que el nodo no contiene soluciones factibles, o se ha identificado la mejor solución factible en el nodo (de tal forma que $L(N)$) corresponde al valor de su función objetivo).

Paso 4: Reconocer cuando un nodo final contiene una solución óptima. El proceso termina cuando no existen nodos restantes (no examinados) y la solución actual es la solución óptima, (si no existe tal solución entonces el problema no tiene soluciones factibles). De otra manera se regresa al paso 1.

Si el objetivo es maximizar solo se invierten los papeles de las cotas y se invierten las direcciones de las desigualdades.

2.2.3 Convergencia del algoritmo

La operación de ramificación del algoritmo garantiza una solución óptima en un número finito de iteraciones. Como T es finito, el proceso de ramificación conducirá eventualmente a una

solución T_i, como nodo final, a menos que se detenga previamente. El proceso de ramificación produce una partición de T para la cual cada subconjunto T_i, obedece, a la definición de partición. Así todos los nodos finales posibles se pueden generar y obtenerse así la solución óptima. Las características de la ramificación prometen una enumeración completa para el algoritmo de R-A a menos que se pueda encontrar una solución óptima antes de hacerlo. Las características de acotamiento del algoritmo de R-A proporcionan la posibilidad de reconocer una solución óptima antes de completar la enumeración.

2.2.4 Estrategias operativas en el algoritmo básico

Existen diferentes opciones de llevar a cabo la operación de ramificación. Se identifican tres formas distintas de ramificar: ordenada, inmediata del sucesor y ramificación de sucesor.

En aras de hacer más sencilla la exposición planteamos un problema de permutación y a través de él se exponen las distintas técnicas de ramificación.

Se desea ordenar un conjunto de n artículos $A=\{a_1,...,a_n\}$, esto es, buscamos un ordenamiento que sea óptimo de los n artículos bajo condiciones dadas. El conjunto de soluciones factibles consiste de vectores n-dimensionales.

$$F=\{X / x_i \in A \quad y \quad x_i \neq x_j \quad si\ i \neq j\}$$

Donde el vector x, representa un arreglo de los n artículos. La exigencia de que xi sea diferente a x_i si i es distinto de j, se debe a que un mismo artículo a_i, no puede ocupar dos lugares en un mismo arreglo.

A continuación se discuten las posibles reglas de ramificación para el caso en que $n=4$ y $A=\{a,b,c,d\}$.

2.2.4.1 Ramificación ordenada

Vamos a llamar x al conjunto de soluciones factibles que contiene los artículos ya seleccionados, llamaremos $R(X)$ al rango de X, es el artículo que vamos a seleccionar para introducir la solución factible que se construye, llamaremos nivel de X, al número de artículos seleccionados, así sea $R=\{r_1,...,r_p\}$ un subconjunto de los índices de los artículos $\{1,2,...,n\}$ que contiene p elementos distintos, que se han introducido al arreglo.

Un nodo X de un árbol de búsqueda se determina al imponer las siguientes restricciones.

$x_{r1}=a_{j1}, x_{r2}=a_{j2},...,x_{rp}=a_{jp}$.

Para algún conjunto $\{a_{j1},...,a_{jp}\}$ de p artículos distintos (los seleccionados), de donde, la siguiente selección de X la haremos sobre los restantes $n-p$ artículos a ordenar, lo que nos genera una partición de X en $n-p$ subconjuntos ajenos, al especificar el artículo que

tenga orden $r(X)$ para alguna $r(X)$ que no está en R, seleccionamos también el nuevo conjunto de soluciones factibles a partir.
La regla de ramificación ordenada nos especifica las reglas explícitas de selección, para asignar un rango a $X : R(X)$, de esta forma $r(x)$ se considera sólo en función del nivel de X (donde p = nivel de X). Por ejemplo: $r(x) = p+1$ consiste en seleccionar primero el artículo que irá en primer lugar del arreglo, después el del segundo lugar y así sucesivamente, de esta forma, si se tiene un conjunto $X = \{x_1, x_2, x_3, x_4\}$ donde la regla de selección es "mayor que" se tiene x_1, x_2, x_3, x_4. Si la regla fuera $r(X) = n - p$ entonces empezaríamos por seleccionar el último del arreglo, y terminar en el primero, en el ejemplo anterior la solución es la misma pero la selección se hace considerando primero a x_4, después a x_3 y así hasta x_1.

La Figura 2.3 muestra un árbol de enumeración completa, para $n = 4$ y $A = \{a, b, c, d\}$ generado por ramificación ordenada.

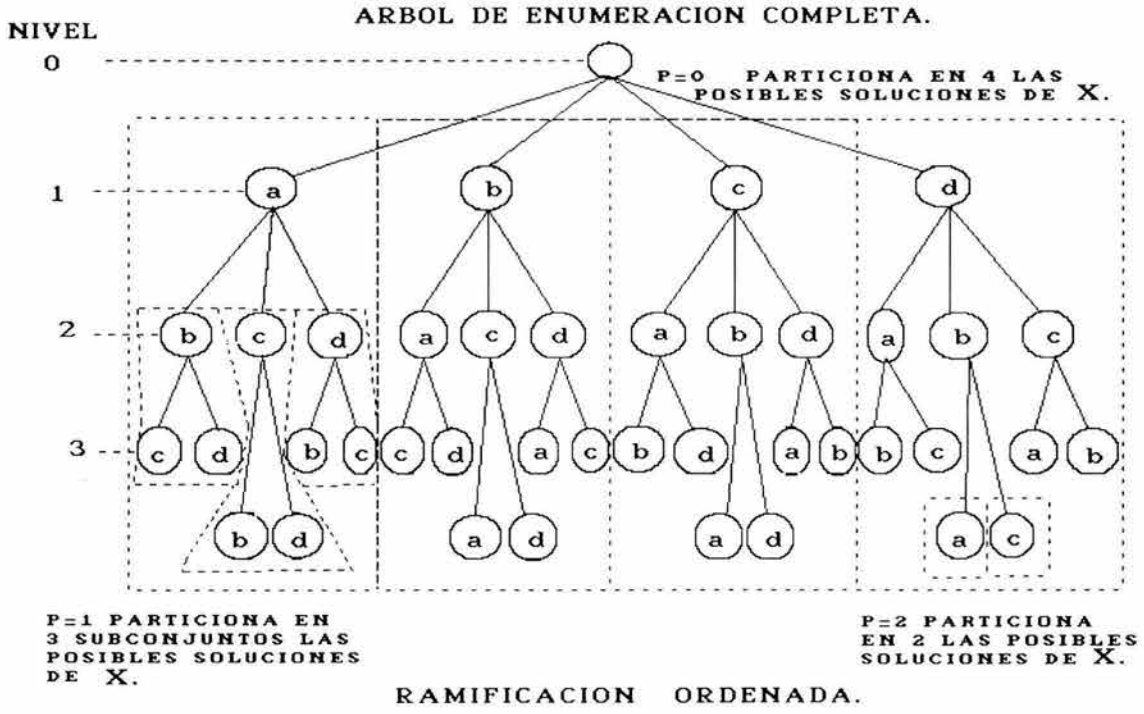


Figura 2.3 Árbol de enumeración completa

Otra regla de selección consiste en ir insertando en forma ordenada cada uno de los artículos de la manera siguiente: al primer artículo le asignamos su lugar en el arreglo solución. Por ejemplo, considere los nodos a, b, c y d en los niveles 0, 1 2 y 3 respectivamente. Entonces Z corresponde a la solución:

$$(b, d, a, c) \;\; si \;\; r(a) = 3, \;\; r(b) = 1, \;\; r(c) = 4, \;\; r(d) = 2$$

(b,a,d,c) *si* $r(a)=2,\ r(b)=1,\ r(c)=4,\ r(d)=3$
(a,b,c,d) *si* $r(x)=(Niveldex)+1 \quad \forall x$
(d,c,b,a) *si* $r(x)=(Niveldex)+1 \quad \forall x$

2.2.4.2 Ramificación inmediata del sucesor

Sea X un nodo del árbol especificado por *p* (= nivel de X) relaciones de la forma $r_j = r_i + 1$ que puede interpretarse como el renglón a_j ordenado inmediatamente después del renglón a_i. Entonces X se puede particionar en dos conjuntos:

$$X(kl) = \{x / x \in X \ \ y \ \ r_l = r_k + 1\}$$
$$X(kl) = X - X(kl) = \{x / x \in X \ \ y \ \ r_l \neq r_k + 1\}$$

Se requiere una regla de selección para decidir el par de índices *k* y *l*. La figura 2.4 da un ejemplo de un árbol de enumeración completa para n=4 y A={a, b ,c, d}.

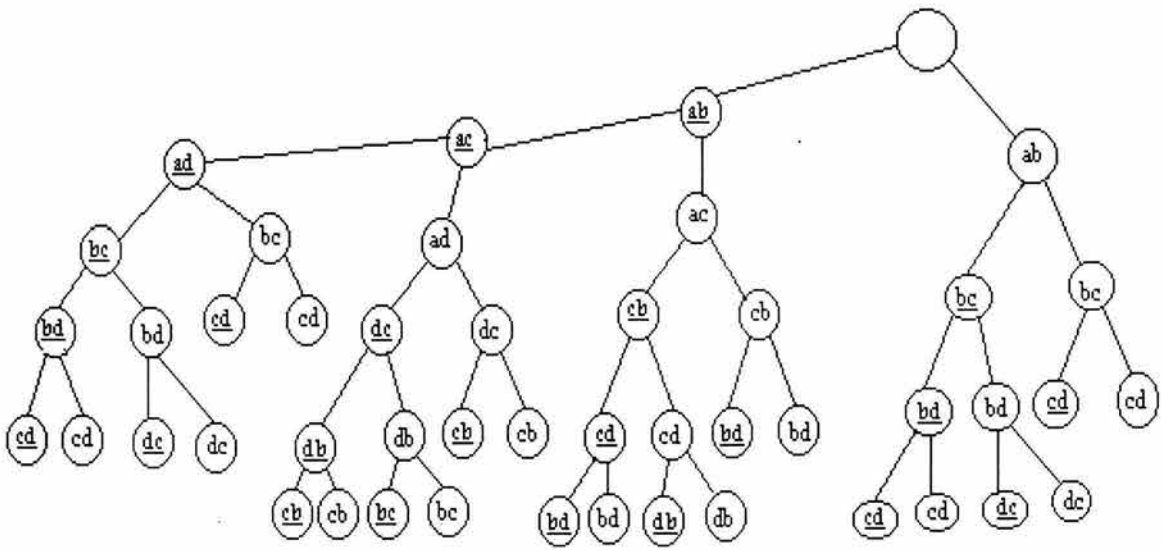


Figura 2.4 Árbol de ramificación inmediata del sucesor

2.2.4.3 Ramificación del sucesor

Sea X un nodo del árbol especificado por p relaciones $r_j \succ r$ (que se interpretan como un renglón a_j ordenado posteriormente a un renglón a_i). Donde X puede particionarse en dos conjuntos:

$$X(kl) = \{x / x \in X \ \ y \ \ r_l = r_k + 1\}$$
$$X(kl) = X - X(kl) = \{x / x \in X \ \ y \ \ r_l \neq r_k + 1\}$$

Nuevamente se requiere de una regla de selección para escoger la pareja $(k,l)\ k \neq 1$. La Figura 2.5 da un ejemplo de un árbol de enumeración completa.

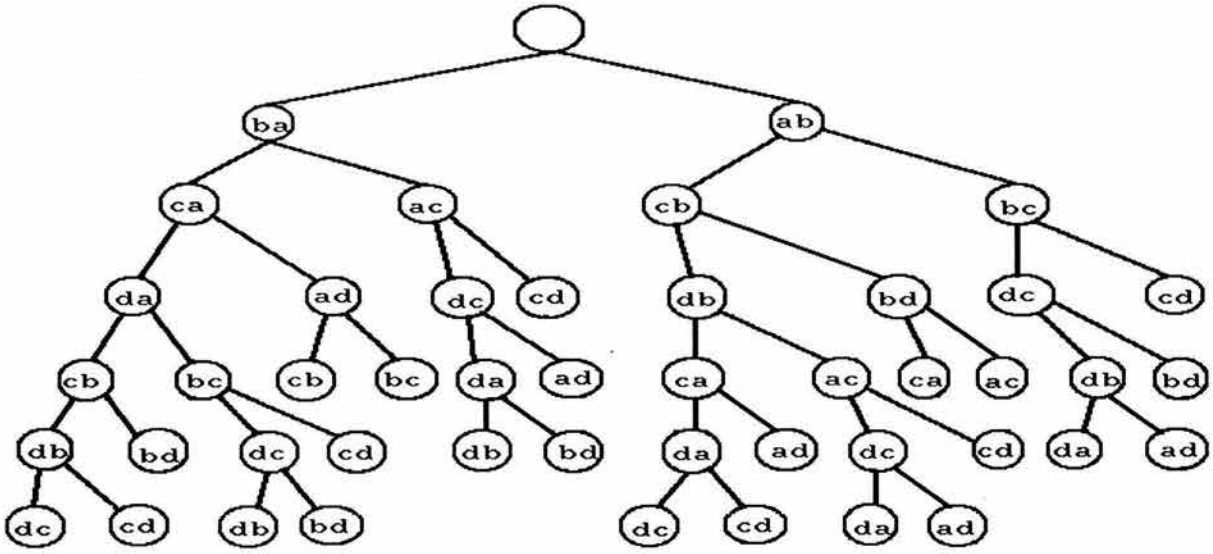


Figura 2.5 Árbol con ramificación del sucesor

2.2.4.4 Acotamiento

Un árbol de búsqueda puede formarse con una pequeña parte de un árbol de enumeración completa (en el cual no se descarta ningún nodo, se consideran todos los posibles), debido al tamaño tan grande de éste. Para poder hacer esto escogemos una regla de acotamiento adecuada. Una función de acotamiento es una función $g: 2^T \rightarrow R$ que asigna un número real $g(x)$ a cada subconjunto T tal que si X está contenido en T entonces la cota $g(x)$ satisface:

$$g(x) \leq f(x) \quad \forall x \in X \bigcap F$$

(Note que las cotas son todas cotas inferiores ya que estamos suponiendo un problema de minimización). Podemos suponer también que

$$g(\{x\}) = f(x) \quad \forall x \in F$$

Cotas para un problema dado *P* pueden obtenerse relajando alguna de las restricciones. Esto da el problema relajado *Q* cuyo conjunto factible F_Q contiene al conjunto factible F_P del problema original *P*. Sea X un subconjunto de T entonces:

Claramente g es una función de acotamiento inferior.

2.2.4.5 Refinamientos

Para encontrar todas las soluciones en un problema determinado, puede hacerse un análisis para acortar el proceso de búsqueda este es llamado método de *Prevención o Poda de ramas* (preclusion o branch pruning), por el efecto de remover subárboles del árbol. Un problema puede podarse aún más y las soluciones pueden ser consideradas como equivalentes si se hace una serie de reflexiones. Es claro que si se encuentran soluciones equivalentes, se puede producir fácilmente un conjunto de todas las soluciones y de esa manera se poda un gran subárbol del árbol. [Castañeda, 2000]

Otro refinamiento de esta técnica es el método de *Fusión u Ordenamiento de Ramas* (fusión o branch merging), donde la idea es evitar hacer dos veces el mismo trabajo: si dos o más subárboles del árbol son isomorfas, se puede buscar solamente en uno de ellos. El hecho de ahorrar con este método no es trivial dado que puede haber un gran ahorro en el número total de nodos que intervienen en la búsqueda.

Además, para el método de fusión u ordenamiento, el cual claramente puede acortar la búsqueda si se busca la solución en el árbol entero, hay una técnica heurística que puede ser útil cuando la existencia de una solución está en duda o cuando solamente se encuentra una solución en lugar de todas las soluciones. Esta técnica, es llamada *árbol de arreglos* (tree arrangement) y se puede usar en varias formas. Primero, si la evidencia indica que todas las soluciones tendrán una forma particular, entonces se poda para estructurar la búsqueda, así que las soluciones potenciales de esa forma se inspeccionan antes que otras soluciones. Segundo; si es posible, el árbol se debe rearreglar, así que los nodos de bajo grado (por ejemplo esos con relativamente pocos hijos) deben estar cerca de la parte de arriba del árbol; por ejemplo, el árbol de la Figura 2.6.a es preferible a uno como el de la Figura 2.6.b.

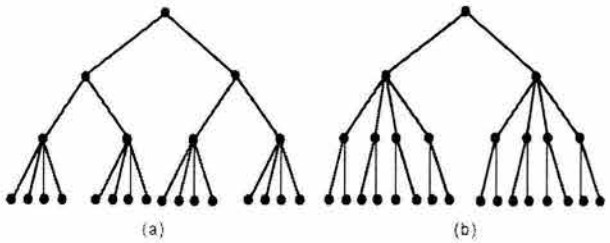


Figura 2.6 Dos posibles árboles para ser examinados con retroceso

De esto nace la pregunta de ¿por qué esto es útil?; generalmente, para descubrir que camino no puede ser tratado para una solución, varias restricciones deben ser acumuladas, y normalmente esto sucede en una profundidad fijada desde la parte de arriba de los árboles. El resultado es que se podarán más nodos si están más cerca de las hojas que de la raíz. Por supuesto, debe entenderse que este tipo de arreglo de árbol puede no ayudar, si se debe buscar en el árbol entero. Más aún, si en el árbol entero no se necesita buscar, la búsqueda en el rearreglo puede mover las soluciones más cerca pero más lejos del inicio de la búsqueda.

Un último refinamiento es la técnica del *problema de descomposición* (problem decomposition), donde se descompone el problema en *k* subproblemas, se resuelven los subproblemas, y entonces se componen las soluciones para los subproblemas en una solución del problema original. La demanda de almacenamiento puede ser alta porque todas

las subsoluciones deben ser guardadas, pero el incremento en la velocidad puede ser considerable. Por ejemplo si toma en tiempo $c2^n$ para resolver el problema de tamaño n, entonces el tiempo es reducido a $kc2^{n/k} + T$, donde T es el tiempo requerido para componer las subsoluciones. Si el número de subsoluciones es pequeño y si no es tan difícil componerlas, entonces T será relativamente pequeño y la técnica dará como resultado un enorme ahorro.
Las variaciones como son la técnica de Búsqueda Exhaustiva Ingenua (Naive Exhaustive Search), la de Ramificación y Acotamiento (Naive Branch and Bound) y la búsqueda de una Mejor Ramificación y Acotamiento (A better Branchand Bound) son un tipo específico de prevención. La prevención está basada en el hecho de que cada una de las soluciones tiene un costo asociado y que se podrá encontrar la solución óptima (ya sea con costo mínimo o máximo).

La tres técnicas de Búsqueda antes mencionadas están basadas en un árbol de búsqueda, donde en cada etapa todas las posibles soluciones del problema son particionadas en dos o más subconjuntos, cada uno representado por nodos en un árbol de decisiones. Esta búsqueda (particionamiento) se realiza de acuerdo con alguna heurística, la cual reduce la búsqueda que conduce a la solución óptima. Después de ramificar, las cotas más pequeñas son computadas sobre el costo de cada uno de los dos subconjuntos.

2.3 Algoritmo de Ramificación y Acotamiento para el agente viajero

Los pasos de ramificación y acotamiento permiten una gran flexibilidad al diseñar un algoritmo específico para un determinado problema y tienen un efecto importante sobre la eficiencia computacional del algoritmo. Es decir, existen varias estrategias de ramificación y acotamiento dependiendo del problema en consideración.

En el caso del problema del viajero, existen tres métodos de ramificación y acotamiento. La diferencia principal entre estos algoritmos es la estrategia de acotamiento que usan.

1. Construcción del circuito según la matriz reducida.
2. Eliminación de subcircuitos a partir de la solución de problemas de asignación.
3. Construir la trayectoria basándose en el árbol de expansión mínima.

A continuación se estudiarán tres algoritmos exactos que usan el método de ramificación y acotamiento para obtener la solución óptima. El primer método que se describirá se basa en la construcción del circuito hamiltoniano según la matriz reducida o bien el algoritmo little-murty, el segundo se basa en la eliminación de subcircuitos a partir de la solución de problemas de asignación y existe un tercer método que se basa en construir la trayectoria basándose en el árbol de expansión mínima. Existen muchos mas algoritmos que utilizan la técnica de ramificación y acotamiento, pero para fines de este trabajo únicamente se consideraron estos tres algoritmos. [Flores, 2000]

2.3.1 Algoritmo little-murty

Propósito: determinar el circuito hamiltoniano de costo mínimo, dada una matriz de costos. El método utiliza la propiedad de la matriz de costos reducida para probar la inclusión o exclusión de un arco en el circuito.

Descripción

Paso 1: Dada la matriz de costos D, se efectúan sustracciones en los renglones y las columnas de la matriz D, sin permitir que aparezcan valores negativos. Con esto obtenemos una cota inferior del problema del viajero al sumar los elementos que se restaron a los renglones y a las columnas. La matriz D' es la matriz reducida de D.

Paso 2: Sea S un nodo del árbol y $ev(S)$ la cota inferior de este nodo S, con cada arco (i, j) con $d_{ij}' = 0$, debemos usar algún arco comenzando en i y penalizarlo por α_i. También debemos usar algún arco en j, y podríamos penalizarlo por β_j con lo cual $\Theta_{ij} = \alpha_i + \beta_j$ es la penalización de no escoger *(i,j)*. Se selecciona el arco que tiene el máximo de los Θ_{ij}

Paso 3 :Si el arco *(i, j)* no se selecciona, $ev(S) + \Theta_{ij}$ es una cota inferior. Si el arco *(i, j)* se seleccionó, entonces la matriz se reduce omitiendo el renglón i y la columna j. Buscar la condición adicional sobre D' para excluir subcircuitos posibles.

Paso 4: Seleccione el vértice que tiene menor costo y vaya al paso 1.

En el ejemplo siguiente se ilustra este algoritmo. [Flores, 2002]

Ejemplo 2.1

Sea la matriz de tiempos D que se presenta en la Tabla 2.1, entonces teniendo el tiempo de duración de cada uno de los viajes entre las distintas paradas, se desea encontrar el circuito hamiltoniano de menor duración.

	A	B	C	D	E	F
A	∞	27	43	16	30	26
B	7	∞	16	1	30	25
C	20	13	∞	35	5	0
D	21	16	25	∞	18	18
E	12	46	27	28	∞	5
F	23	5	5	9	5	∞

Tabla 2.1

Paso 1

Si T es la duración de un recorrido hamiltoniano asociado a la matriz de tiempos de la Tabla 2.1, entonces la duración de ese mismo circuito con la matriz de tiempos obtenida de restar un escalar h_i al renglón i está dado por $T - h_i$ porque cada circuito contiene uno y solo un elemento de este renglón. Lo mismo sucede si restamos un escalar h^j de una columna j (j = A, B, C, D, E, F). Una cota inferior del circuito hamiltoniano óptimo es sencilla de obtener si restamos de cada renglón de la matriz de tiempos D, el mínimo elemento correspondiente. O bien

$$h^j = \min_j d_{ij}^1 \geq 0$$

	A	B	C	D	E	F
A	∞	11	27	0	14	10
B	6	∞	15	0	29	24
C	20	13	∞	35	5	0
D	5	0	9	∞	2	2
E	7	41	22	23	∞	0
F	18	0	0	4	0	∞

Tabla 2.2

Y entonces, en la matriz D', así reducida, restamos de cada columna j la constante

$$h^j = \min_j d_{ij}^1 \geq 0$$

La nueva matriz D^1 tiene elementos no-negativos y contiene al menos un elemento cero en cada renglón y cada columna. Como se muestra en la Tabla 2.3

	A	B	C	D	E	F
A	∞	11	27	0	14	10
B	6	∞	15	0	29	24
C	20	13	∞	35	5	0
D	5	0	9	∞	2	2
E	7	41	22	23	∞	0
F	18	0	0	4	0	∞

Tabla 2.3

Tomamos:

$$h = \sum_i h_i + \sum_j h_j$$

La duración $z(t)$ de un circuito hamiltoniano t de la matriz D es entonces igual a $z(t) = h + z'(t)$, donde $z'(t)$ es la duración del circuito t de la matríz D'. Ya que $z'(t)$ (porque $d_{ij}^1 \geq 0$), entonces tenemos $z(t) \geq h$, donde h es una evaluación por cota inferior del conjunto Ω de circuitos hamiltonianos y se denota $ev(\Omega)$.

De la matriz dada D^1 obtenemos:

$$h_1 = 16,\ h_2 = 1,\ h_3 = 0,\ h_4 = 16,\ h_5 = 5,\ h_6 = 5$$
$$h^1 = 5,\ h^2 = h^3 = h^4 = h^5 = h^6 = 0$$

Un aspecto importante es que cada circuito hamiltoniano asociado a la Tabla 2.3 tiene una duración no negativa y que difiere en tiempo del circuito original en 48 unidades de tiempo. De donde una cota inferior de la duración del circuito mínimo es 48 o bien:

$$ev(\Omega) = \sum h_i + \sum h^j$$

Paso 2

Considerando todas las entradas de la matriz de la Tabla 2.3 iguales a cero, calculamos los retardos o penalizaciones por no usar alguno de esos arcos que prometen un circuito de menor costo, ya que fueron los que al restar los mínimos en renglones y columnas quedaron igual a cero.

Calculamos el retardo Θ_{ij} correspondiente a $d_{ij}' = 0$

$$\begin{aligned}
\Theta AD &= \alpha A + ßD = 10 + 0 = 10 \\
\Theta BD &= \alpha B + ßD = 1 + 0 = 1 \\
\Theta CF &= \alpha C + ßF = 5 + 0 = 5 \\
\Theta DA &= \alpha D + ßA = 0 + 1 = 1 \\
\Theta DB &= \alpha D + ßB = 0 + 0 = 0 \\
\Theta EF &= \alpha E + ßF = 2 + 0 = 2 \\
\Theta FB &= \alpha F + ßB = 0 + 0 = 0 \\
\Theta FC &= \alpha F + ßC = 0 + 9 = 9 \\
\Theta FE &= \alpha F + ßE = 0 + 2 = 2
\end{aligned}$$

En general, un arco *(i,j)* con $d_{ij}' = 0$ no se escoge, ya que debemos dejar el punto *i*, y usar algún arco comenzando en *i* y penalizarlo por αi. También debemos usar algún arco en *j*, y podríamos penalizarlo por ß*j*, con lo cual $\Theta_{ij} = \alpha i + \beta j$ es la penalización de no escoger *(i, j)*. Dicha penalización Θ_{ij} se puede interpretar en este caso como un retardo.

Si $ev(S)$ es la evaluación obtenida reduciendo la matriz por el nodo *S*, entonces $ev(S) + \Theta_{ij}$ es una primer evaluación por cota inferior del nodo *ij*, obtenido de *S* por no escoger el arco *(i, j)*.

Paso 3

Separamos la pareja *(i, j)* que maximiza el retardo Θ_{ij} para garantizar una mayor cota inferior de la duración de los circuitos hamiltonianos $\Theta AD = 10$.

Entonces el nodo *AD* tiene una evaluación por cota inferior igual a 48+10=58

Una manera de proceder a la determinación del circuito hamiltoniano de mínima duración consiste en particionar el conjunto de circuitos hamiltonianos como sigue:

a) Circuitos hamiltonianos que usan el arco *AD*
b) Circuitos hamiltonianos que no usan el arco *AD*

En el primer caso la matriz de tiempos entre localidades se reduce a una nueva matriz en donde se elimina el renglón *A* y la columna *D*. Asimismo, el tiempo entre la localidad *D* a la *A* se hace igual a infinito o a un número muy grande para evitar usar el arco *DA*; pues sabemos que no forma parte del circuito hamiltoniano mínimo. Si no hacemos este tiempo infinito,

existe la posibilidad de la aparición de subcircuitos. La Tabla 2.4 muestra la matriz reducida donde se han eliminado el renglón *A* y la columna *D*, el arco *DA* se hace igual a infinito.

	A	B	C	E	F
B	1	∞	15	2 9	24
C	15	13	∞	5	0
D	∞	0	9	2	2
E	2	41	22	∞	0
	13	0	0	0	∞

Tabla 2.4

Una cota inferior de la duración de los circuitos hamiltonianos asociados con esta Tabla es sencilla de obtener, si restamos una unidad a cada elemento del renglón uno, como se muestra en la Tabla 2.5

	A	B	C	E	F
B	0	∞	14	28	23
C	15	13	∞	5	0
D	∞	0	9	2	2
E	2	41	22	∞	0
F	13	0	0	0	∞

Tabla 2.5

Paso 4

Cabe hacer notar que, debido al resultado de las manipulaciones anteriores, podemos decir que los circuitos hamiltonianos asociados a la Tabla 2.1, que usen el arco *AD* , tienen una duración no menor de 49 unidades de tiempo, 48 acumuladas hasta la obtención de la tabla 2.3 y una unidad de tiempo al pasar de la Tabla 2.4 a la Tabla 2.5, por lo que 49 unidades representan una cota inferior de la duración de los circuitos hamiltonianos que usan el arco *AD*.

Paso 2

El siguiente paso consiste en calcular los retardos correspondientes a la Tabla 2.5

$$\Theta BA = \alpha B + ßA = 14 + 2 = 16$$
$$\Theta CF = \alpha C + ßF = 5 + 0 = 5$$
$$\Theta DB = \alpha D + ßB = 2 + 0 = 2$$
$$\Theta EF = \alpha E + ßF = 2 + 0 = 2$$
$$\Theta FB = \alpha F + ßB = 0 + 0 = 0$$
$$\Theta FC = \alpha F + ßC = 0 + 9 = 9$$
$$\Theta FE = \alpha F + ßE = 0 + 2 = 2$$

Se escoge el arco *BA* para efectuar la ramificación como sigue:

a) Circuitos hamiltonianos que usan el arco *BA*

b) Circuitos hamiltonianos que no usan el arco *BA*

Paso 3

En el primer caso partimos de la Tabla 2.5 eliminando el renglón *B* y la columna *A* y haciendo que el tiempo de *A* a *B* sea igual a infinito para evitar circuitos innecesarios. En este momento, como *AD* y *BA* se han seleccionado para formar el circuito hamiltoniano hacemos que el tiempo de *DB* sea infinito, la Tabla 2.6 exhibe esta situación

	B	C	E	F
C	13	∞	5	0
D	∞	9	2	2
E	41	22	∞	0
F	0	0	0	∞

Tabla 2.6

Con el propósito de tener un elemento cero en cada renglón y columna, así como mejorar la cota inferior de los circuitos hamiltonianos, procedemos a restar dos unidades del renglón dos en la Tabla 2.6 para obtener (Tabla 2.7)

	B	C	E	F
C	13	∞	5	0
D	0	7	0	2
E	41	22	∞	0
F	0	0	0	∞

Tabla 2.7

Y se puede observar que una cota inferior de los circuitos hamiltonianos que usan *BA* es 51 unidades de tiempo.

Paso 2

Nuevamente calculamos las penalizaciones para saber qué arco es el que se va a usar:

$$\Theta CF = \alpha C + ßF = 5 + 0 = 5$$
$$\Theta DE = \alpha D + ßE = 0 + 0 = 0$$
$$\Theta DF = \alpha D + ßF = 0 + 0 = 0$$
$$\Theta EF = \alpha E + ßF = 22 + 0 = 22$$
$$\Theta FB = \alpha F + ßB = 0 + 13 = 13$$
$$\Theta FC = \alpha F + ßC = 0 + 9 = 9$$
$$\Theta FE = \alpha F + ßE = 0 + 2 = 2$$

Paso 3

Como se tiene 22 para EF entonces se incluye al arco EF y si no se incluye se tiene una penalización de 51 + 22 = 73 y la tabla 2.8 exhibe el resultado:

	B	C	E
C	13	∞	5
D	∞	7	0
F	0	0	0

Tabla 2.8

Nos fijamos en el primer renglón y restamos 5 para tener cero quedando:

	B	C	E
C	8	∞	0
D	∞	7	0
F	0	0	0

Tabla 2.9

Paso 2

Lo que nos da un costo de 56 unidades nuevamente calculamos las penalizaciones:

$$\Theta CE = \alpha C + ßE = 8 + 0 = 8$$
$$\Theta DE = \alpha D + ßE = 7 + 0 = 7$$
$$\Theta FB = \alpha F + ßB = 0 + 8 = 8$$
$$\Theta FC = \alpha F + ßC = 0 + 7 = 7$$

Paso 3

De acuerdo con esto podemos elegir como arco a usar el *CE* o el *FB*, escogemos *FB* y obtenemos la Tabla 2.10

	C	E
C	∞	0
D	7	0

Tabla 2.10

Restamos siete en el renglón de *D* lo que nos da un costo de 63 unidades por usar el arco *FB* y obtenemos:

	C	E
C	∞	0
D	0	0

Tabla 2.11

De donde los únicos arcos que quedan que se pueden usar son *CE*, *DC* y *DE* ,de donde escogemos *CE* y *DC* y asignamos infinito a *DE* para evitar posibles subcircuitos con lo que nos queda la Tabla 2.12

	C	E
C	∞	0
D	0	∞

Tabla 2.12

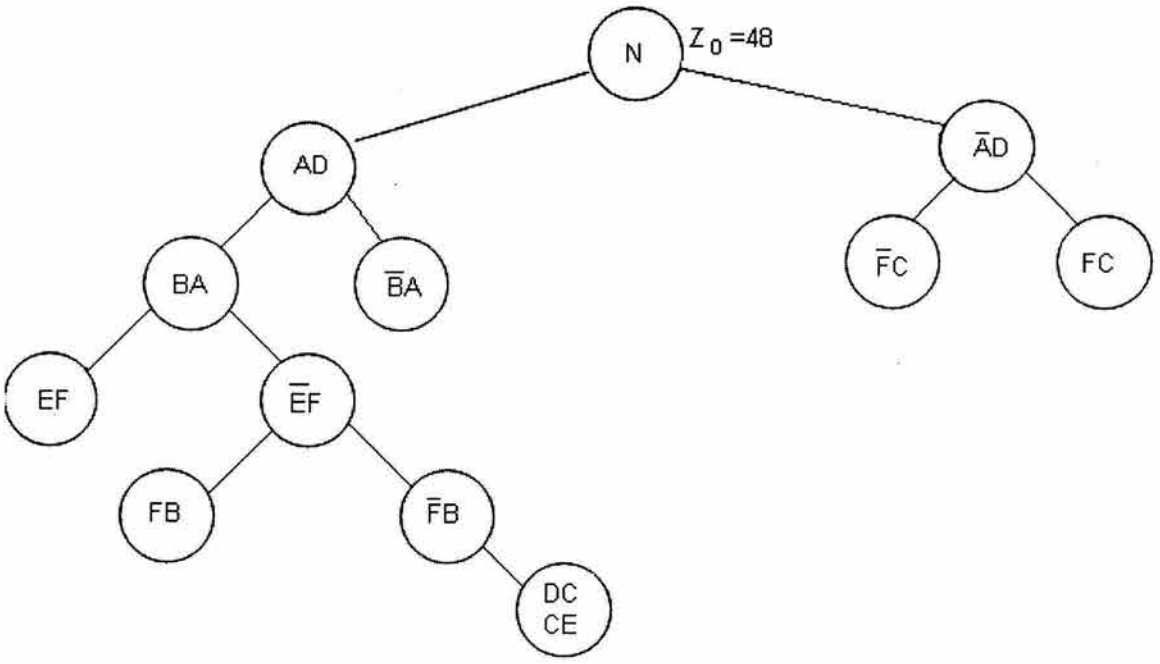


Figura 2.7 Solución gráfica por medio del algoritmo little-murty

De donde se concluye que el circuito hamiltoniano óptimo es *A-D-C-E-F-B-A* con una duración de 63 unidades de tiempo.

Para verificar que esta solución es óptima, debemos examinar el vértice *AD* cuya evaluación por cota inferior es 58 (< 63). La separación del vértice *AD* produce: usando FC una cota de 63 y sin usar FC, una cota de 67, por lo cual el circuito propuesto es óptimo.

2.3.2 Algoritmo basado en el problema de asignación

El problema del agente viajero se puede escribir como un modelo de asignación de la siguiente manera:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij}$$

$$\sum_{j=1}^{n} x_{ij}=1 \qquad i=1,...,n$$

$$\sum_{i=1}^{n} x_{ij}=1 \qquad j=1,...,n$$

$$\sum_{i=1}^{n} x_{ij}=0 \text{ o } 1$$

$$x=x_{ij}$$

Donde $c_{ij}=\infty$ para $i=j$, y $x_{ij}=1$ si el agente viajero va de la ciudad i a la ciudad j.

La relajación más sencilla para el problema del agente viajero y también la primera en ser usada es la que se obtiene de la formulación anterior al eliminar la restricción de que $x=(x_{ij})$ debe ser la asignación de un *tour*. Es decir, el problema de asignación con la misma función de costos que la del problema del agente viajero, el cual se resuelve eficientemente mediante el *método húngaro*. Esta es la estrategia de acotamiento que se usará.

2.3.2.1 Estrategia de acotamiento

La aplicación del método de R-A tiene diferentes formas dependiendo de las operaciones de ramificación, pero la operación de acotamiento siempre es la misma y está basada en la solución óptima del problema de asignación bajo la matriz de costos supuesta.

Las tres maneras diferentes de efectuar la operación de ramificación son:

a) Ramificación simple: Romper el subcircuito de la solución del problema de asignación suponiendo que uno de los arcos del subcircuito tiene costo infinito.

b) Ramificación disjunta: Suponer que uno de los arcos del subcircuito tiene costo infinito pero un subconjunto de los arcos del subcircuito tiene que estar conectado forzosamente.

c) Ramificación mejorada: Romper el subcircuito suponiendo que todos los arcos tienen consto infinito excepto uno de ellos.

Considérese un *ejemplo* del problema del agente viajero con matriz de costos C. Se resuelve el problema relajado; es decir, el problema de asignación de costos C, mediante el *método húngaro*. Sea $\overline{C}$ la matriz de costos reducida que se obtiene y sea r_0 el valor óptimo de este problema. Recuérdese que la matriz que se obtiene de la matriz original de costos al restarle una constante a todas las entradas de un renglón o una columna, se le conoce como matriz de costos reducida. Entonces, para cualquier asignación de un *tour* x se tiene que:

$$Z_C(x) = r_0 + Z_{\bar{C}}(x) \qquad (2.1)$$

Ahora, se sabe que la matriz $\bar{C}$, es no negativa. Entonces,

$$r_0 \leq Z_C(x) \quad \textit{para toda x asignacion de un tour} \qquad (2.2)$$

Por lo tanto r_0 es una cota inferior para el problema original del agente Viajero.

Supóngase que la asignación óptima x^* es la asignación óptima de un *tour*, por lo tanto el valor de x^* en la matriz de costos reducida $\bar{C}$ es cero.

Entonces:

$$Z_C(x^*) = r_0 + Z_{\bar{C}}(x^*)$$
$$\Rightarrow Z_C(x^*) = r_0 + 0$$
$$\Rightarrow Z_C(x^*) = r_0$$

De aquí que la igualdad (2.2) se da cuando la asignación óptima es la asignación de un *tour*. En este caso, ésta es la solución óptima para el problema del agente viajero.

2.3.2.2 Estrategias de ramificación

El problema del agente viajero es un problema binario, por esta razón se usará un procedimiento de enumeración implícita para resolverlo. Recuérdese que es este tipo de procedimientos se empieza con una o más variables que se fijan en un valor binario y gradualmente se construye las solución al aumentar el número de variables fijas. Entonces, la regla de ramificación selecciona alguna variable libre x_{rs} en uno y en cero. Al ramificar el problema original (subproblema 1) se generan los siguientes subproblemas:

$$\textit{subproblema } 2 = \textit{subproblema } 1 + (x_{rs} = 0)$$
$$\textit{subproblema } 3 = \textit{subproblema } 1 + (x_{rs} = 1)$$

Si se elige el subproblema 2 para acotarlo inferiormente se debe resolver el problema de asignación cuya matriz de costos es C_1. Esta matriz se obtiene a partir de la matriz $\bar{C}$ al modificar la entrada $\bar{C}(r,s)$ por M, donde $M \geq 0$, ya que si $x_{rs} = 0$ no se puede viajar de la ciudad r a la ciudad s. Por otro lado, sea $\underline{C}$ la matriz que se obtiene a partir de la matriz original C al modificar la entrada $c(r,s)$ por M. Entonces, para cualquier asignación x de un *tour* se tiene que:

$$Z_{\bar{C}}(x) = r_0 + Z_{C_1}(x)$$

Supóngase que el valor óptimo del problema de asignación asociado con el subproblema 2 es r_1 y sea $\overline{C_1}$ la matriz de costos reducida. Entonces, para cualquier asignación de un *tour* x se tiene que:

$$Z_{C_1}(x) = r_1 + Z_{\overline{C_1}}(x)$$

Pero,

$$Z_{\overline{C}}(x) = r_0 + Z_{\overline{C_1}}(x) \text{ para toda } x \text{ asignacion de un tour}$$
$$\Rightarrow Z_{\overline{C}}(x) = r_0 + r_1 + Z_{\overline{C_1}}(x) \text{ para toda } x \text{ asignacion de un tour}$$

Además, se sabe que $\overline{C_1} \geq 0$, por lo que,

$$r_0 + r_1 \leq Z_{\overline{C}}(x) \text{ para toda } x \text{ asignacion de un tour}$$

Entonces $r_0 + r_1$ es una cota inferior para el valor objetivo de cualquier asignación de un *tour* con matriz de costos $\underline{C}$. Es decir, $r_0 + r_1$ es una cota inferior para el subproblema 2.

Se sabe que una cota inferior para el valor óptimo del problema original es r_0. También se calculó una cota inferior para el valor óptimo del subproblema 2 , $r_0 + r_1$. Entonces r_1, es la cantidad por la cual la cota inferior del subproblema 2 es mayor que la cota inferior para el problema original. Se desea seleccionar la variable de ramificación de tal manera que la cantidad r_1 sea lo más grande posible. Si se selecciona de esta manera la variable de ramificación, entonces la cota inferior para al menos uno de los subproblemas generados será lo más grande posible.

Sin embargo, para determinar la cantidad r_1 de cada variable es necesario resolver un problema de asignación. Esto requiere de muchos cálculos y esto lo hace ineficiente. Entonces, es necesario desarrollar un criterio para seleccionar la variable de ramificación que no involucre muchos cálculos que permita una elección razonable.

Para ello se define la *evaluación de una variable* con respecto a la matriz de costos reducida $\overline{C}$. Para cada variable x_{ij} se calcula:

$$P_{ij} = \min\left\{\overline{C}_{ih};\quad h \neq j\right\} + \min\left\{\overline{C}_{hj};\quad h \neq j\right\}$$

P_{ij} es la mínima cantidad por la cual el valor óptimo del subproblema se incrementa si la variable x_{ij} se fija en cero. La cantidad r_1 es mayor o igual que la evaluación de cada variable. Entonces, una buena regla para seleccionar la variable de ramificación es elegir aquélla cuya evaluación sea la más alta. Por lo tanto, se elige la variable de ramificación x_{rs} tal que:

$$P_{rs} = \max\left\{P_{ij}\right\}$$

Obsérvese que para aquellas variables x_{ij} donde $c_{ij} \neq 0$ la evaluación de la variable P_{ij} es cero, ya que en la matriz reducida $\overline{C}$ al menos hay un cero en cada renglón y en cada columna. Entonces, basta calcular la evaluación de aquellas variables x_{ij} donde $\overline{c}_{ij} = 0$.

Por último, cabe hacer notar que si se establece la variable x_{rs} en uno, para evitar un *subtour* entre las ciudades r y s, la variable x_{rs} debe fijarse en cero.

2.3.3 Descripción del algoritmo basado en el problema de asignación

En cada iteración t de este algoritmo se realizan los siguientes pasos:

1. Se inicializa la cota superior $Z_{UB} : Z_{UB} = \infty$
2. Si faltan nodos del árbol por analizar se elige alguno con la regla de la cota más reciente o con la estrategia de prioridad. Por el contrario, si todos los nodos están analizados el algoritmo termina en esta etapa. Si en este momento se tiene una solución candidata, ésta es la solución óptima del problema original. De lo contrario el problema es infactible.
3. Se acota inferiormente el subproblema seleccionado al resolver el problema de asignación asociado con este subproblema. Sea Z^t el valor óptimo del problema de asignación. Si la asignación óptima corresponde a la asignación de un *tour* entonces ésta constituye una solución candidata. En este caso, se actualiza el valor de la cota superior , $Z_{UB} = Z^t$. Regresar al paso 2. Si la asignación óptima no es factible para el problema original entonces ir al paso 1.
4. Se ramifica el subproblema sobre alguna variable libre x_{rs} generando de esta manera dos subproblemas, uno con $x_{rs} = 0$ y otro con $x_{rs} = 1$ y $x_{sr} = 0$. La variable de ramificación es la que se considera como la evaluación sea la más alta. Regresar al paso2.

Ejemplo 2.2

Considérese el siguiente problema del agente viajero con la siguiente matriz de costos:

$$\begin{pmatrix} i/j & 1 & 2 & 3 & 4 & 5 \\ 1 & M & 1 & 2 & 2 & 3 \\ 2 & 1 & M & 2 & 3 & M \\ 3 & 2 & 2 & M & 2 & 2 \\ 4 & 2 & 3 & 2 & M & M \\ 5 & 3 & M & 2 & M & M \end{pmatrix}$$

Iteración 1

Se inicializa la cota superior Z_{UB} en ∞. Se resuelve el problema relajado de asignación correspondiente al problema original. La matriz de costos reducida $\bar{C}$ que se obtiene es la siguiente:

$$\begin{pmatrix} & 1 & 2 & 3 & 4 & 5 \\ 1 & M & 0 & 2 & 0 & 1 \\ 2 & 0 & M & 1 & 0 & M \\ 3 & 2 & 1 & M & 0 & 0 \\ 4 & 0 & 0 & 0 & M & M \\ 5 & 1 & M & 0 & M & M \end{pmatrix}$$

La asignación óptima es 1-2-4-1,3-5-3 donde $Z^1 = 10$. Dado que esta solución tiene subtuours se debe seleccionar una variable para ramificar el problema. Para ello se calcula la evaluación de aquellas x_{ij} tales que $\bar{c}_{ij} = 0$.

$\text{var}\,iable\ x_{ij}$	$evaluacion\ P_{ij}$
$x_{12}, x_{14}, x_{21}, x_{24}, x_{34}, x_{42}, x_{43}$	0
x_{35}, x_{53}	1

Donde P_{35} y $P_{53} = Max\{P_{ij}\}$. Debido a que hay un empate se elige arbitrariamente alguna de estas dos variables, por ejemplo x_{35}. Al fijar x_{35} en uno y en cero se generan los siguientes subproblemas:

$$subproblema\ 1 = subproblema\ 1 + (x_{35} = 0)$$
$$subproblema\ 3 = subproblema\ 1 + (x_{35} = 1\ \ y\ \ x_{53} = 0)$$

Estos subproblemas se incluyen en la lista de subprobemas no insondeables.

Iteración 2

La lista de nodos no insondeables está formada por los subproblemas 2 y 3. En este ejemplo se usará la regla de la cota más reciente para seleccionar un subproblema. Los dos subproblemas de la lista se generaron al mismo tiempo, entonces se elige uno arbitrariamente, por ejemplo el subroblema 2. La matiriz de costos de este subproblema es:

$$\begin{pmatrix} & 1 & 2 & 3 & 4 & 5 \\ 1 & M & 0 & 2 & 0 & 1 \\ 2 & 0 & M & 1 & 0 & M \\ 3 & 2 & 1 & M & 0 & M \\ 4 & 0 & 0 & 0 & M & M \\ 5 & 1 & M & 0 & M & M \end{pmatrix}$$

La matriz de costos reducida que se obtiene al resolver el problema de asignación es:

$$\begin{pmatrix} & 1 & 2 & 3 & 4 & 5 \\ 1 & M & 0 & 2 & 0 & 0 \\ 2 & 0 & M & 1 & 0 & M \\ 3 & 2 & 1 & M & 0 & M \\ 4 & 0 & 0 & 0 & M & M \\ 5 & 1 & M & 0 & M & M \end{pmatrix}$$

La asignación óptima es 1-5-3-4-2-1 donde $Z^2 = 11$. Esta asignación es la asignación de un "tour", por lo tanto ésta constituye una solución candidata.

La cota superior para el valor óptimo del problema original es $Z_{UB} = 11$.

Iteración 3:

El único nodo no insondeable de la lista corresponde al subproblema 3. La matriz de costos de este subproblema es la siguiente:

$$\begin{pmatrix} & 1 & 2 & 3 & 4 \\ 1 & M & 0 & 2 & 0 \\ 2 & 0 & M & 1 & 0 \\ 4 & 0 & 0 & 0 & M \\ 5 & 1 & M & 0 & M \end{pmatrix}$$

Obsérvese que se ha eliminado el tercer renglón y la quinta columna debido a que la variable x_{35} está fija en uno. La matriz de costos reducida para este subproblema es:

$$\begin{pmatrix} & 1 & 2 & 3 & 4 \\ 1 & M & 0 & 2 & 0 \\ 2 & 0 & M & 1 & 0 \\ 4 & 0 & 0 & 0 & M \\ 4 & 0 & M & 0 & M \end{pmatrix}$$

La asignación óptima es 1-2-4-3-5-1 donde $Z^3 = 11$. Esta asignación corresponde a la asignación de un "tour" y de hecho coincide con la solución del subproblema 2, debido a que se trata de un subproblema simétrico. Por lo tanto ésta es la solución óptima para el problema original y el valor óptimo es $Z^* = 11$. El árbol de ramificación y acotamiento generado es el siguiente:

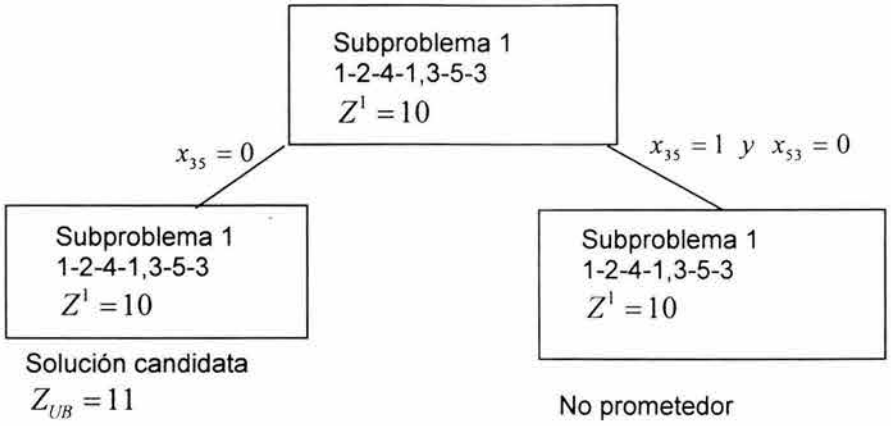


Figura 2.5

2.3.4 Método de R-A basado en el árbol de expansión mínimo

En ciertos casos del problema del agente viajero, se necesita obtener una trayectoria hamiltoniana con costo mínimo. En otras palabras, una trayectoria que pase todos los nodos sin cerrarse. Para la solución de este problema se puede aprovechar la propiedad del árbol de expansión mínimo, en donde el costo total del árbol de expansión mínimo es una cota inferior del problema del agente viajero. Se observa que en el árbol de expansión mínimo puede haber más de dos arcos incidentes en un nodo, lo que pretende este algoritmo es eliminar estos nodos.

Propósito: Dada la matriz de costos simétricos, se encuentra la trayectoria hamiltoniana entre cualquier par de nodos. El nodo inicial y el final no son especificados. El algoritmo aprovecha la conectividad de un árbol y trata de transformarlo a una trayectoria hamiltoniana mediante la toma de decisiones sobre los arcos conectados al nodo que tiene grado mayor que dos.

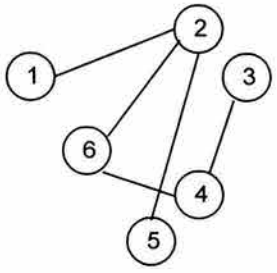
Descripción: La aplicación del método de R-A en este caso consiste en efectuar la opración de acotamiento por medio de la solución del árbol de expansión mínimo de la matriz de costos actual. La operación de ramificación es: suponer cada uno de los arcos excesos en un nodo tienen el costo infinito, de esta manera generar varias alternativas.

Ejemplo 2.3: Considere la matriz de costos siguiente y resuelva usando el método anterior.

De/al	V_1	V_2	V_3	V_4	V_5	V_6
V_1	∞	4	10	18	5	10
V_2	4	∞	12	8	2	6
V_3	10	12	∞	4	18	16
V_4	18	8	4	∞	14	6
V_5	5	2	18	14	∞	16
V_6	10	6	16	6	16	∞

Tabla 2.13

El árbol de expansión mínimo genera la solución ct=22, observemos d(2)=3>2.



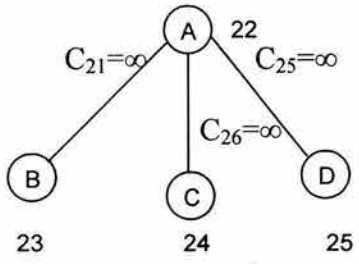
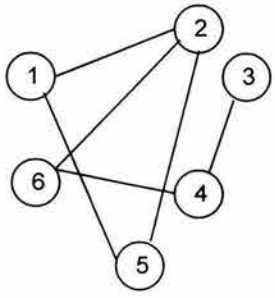
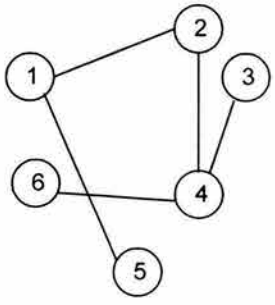
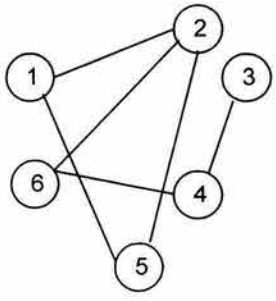


Figura 2.8 a) árbol de expansión mínimo bajo la matriz de costo actual b) árbol de ramificación

Las tres alternativas se muestran como sigue:







Costo actual: 23 (solución óptima) Costo actual: 24 Costo actual: 25

2.4 Método de Búsqueda Exhaustiva Ingenua

El algoritmo de Búsqueda Exhaustiva Ingenua consta de un conjunto N de puntos llamados nodos, éstos son: [Castañeda, 2000]

$$N = \{1, 2, ..., n\}$$

Donde n es el número total de nodos en el grafo (número de vértices o número de ciudades), y cuyo objetivo será generar todas las posibles soluciones y tomar las más cortas.

S es un conjunto ordenado de nodos el cual incluye un camino parcial (que contiene una lista ordenada de k enteros) y la suma de los pesos de sus arcos.

$$S = (k, [i1, i2, ..., ik], peso)$$

Donde $W[i, j]$ es la matriz de adyacencia o de costos que representa un problema PAV, un ejemplo de una matriz de adyacencia para cuatro ciudades se muestra a continuación:

De/al	V_1	V_2	V_3	V_4
V_1	0	3	5	12
V_2	3	0	6	4
V_3	5	7	0	2
V_4	8	1	4	0

Tabla 2.13 Ejemplo de PAV de cuatro ciudades

Se genera un circuito y se compara el peso de ese circuito con el peso del circuito inicial que recibe el nombre de Best_S_soFar. Si el circuito recién calculado resulta ser mejor, entonces este es nuestro nuevo conjunto Best_S_soFar, de lo contrario se tiene el anterior. El proceso se detiene cuando ya ningún circuito mejora en peso al del conjunto Best_S_so_Far.

Los procedimientos que realizan la tarea de búsqueda exhaustiva son dos. El primero de ellos descrito en la Figura 2.9(a) que se llama Búsqueda_Exhaustiva, este es el encargado entre otras cosas de calcular el peso inicial y de llamar a Búsqueda_Exhaustiva_Rec. En la Figura 2.9(b) Búsqueda_Exhaustiva_Rec recursivamente va generando el árbol de búsquedas hasta encontrar el costo mínimo.
Este algoritmo busca todos los $(n - 1)!$ posibles caminos iniciando en 1, guardando el mejor. Hay mucho de paralelismo, porque después de *k* recursivos llamados a Búsqueda_Exhaustiva_Rec, hay $(n - 1)*(n - 2)*...*(n - k)$ subárboles independientes para buscar, el cual puede ser realizado en muchos procesadores. Ya que los subárboles son igualmente largos, se dice que el balance de carga es perfecto. El hecho de que se realice una búsqueda de todos los $(n-1)!$ distintos caminos en el peor de los casos implica que tiene un $O(n!)$.

```
{01} Procedure Búsqueda_Exhaustiva_Ingenua
{02} peso = w(1, 2) + w(2, 3) + w(3, 4) + ..... + w(n-1, n) + w(n, 1);
{03} Best_S_so_far = ( n, [ 1, 2, 3, ....., n-1, n ], peso );
{04} S = ( 1, [ 1 ], 0 );
{05} Búsqueda_Exhaustiva_Rec ( S, Best_S_so_far);
{06} Imprime_Resultados ( Best_S_so_far);
{07} end de Búsqueda_Exhaustiva_Ingenua;
```

Figura 2.9 (a) Algoritmo que llama a Búsqueda_Exhaustiva_Rec

```
{08} Procedure Búsqueda_Exhaustiva_Rec ( S, Best_S_so_far )
{09} Let S = ( k, [ i1, i2, ....., ik ], peso );
{10} Let Best_S_so_far = ( n, [ i1B, i2B, ....., inB ], pesoB );
{11} If k = n
{12} then
{13} new_peso = peso + A(ik, i1);
{14} if new_peso < pesoB
{15} then
{16} Best_S_so_far = ( k, [ i1, i2, ....., ik ], new_peso );
{17} end if;
{18} else
{19} for all j not in [ i1, i2, ....., ik ];
{20} new_peso = peso + A(ik, j);
{21} New_S = ( k + 1, [ i1, i2, ....., ik, j ], new_peso );
{22} Búsqueda_Exhaustiva_Rec ( New_S, Best_S_so_far );
{23} end for;
{24} end if;
{25} return
{26} end de Búsqueda_Exhaustiva_Rec;
```

Figura 2.9 (b) Algoritmo recursivo de Búsqueda Exhaustiva Ingenua

En todos los algoritmos se numerarán las líneas en orden ascendente para poder explicar en caso necesario alguna de ellas haciendo referencia al número. Así, en la Figura 2.9 :

- {02} Calcular el valor inicial del peso *peso* , donde el peso depende del recorrido inicial que puede ser uno escogido arbitrariamente, así que en este método se escogió $\{1,2,3,4, \ldots n\}$, el cálculo del peso matemáticamente se representa como:

$$peso = A(n,1) + \sum_{i=1}^{m=n-1} A(i,i+1)$$

  se agrega el valor $A(n,1)$ para que pueda cerrarse el circuito y su peso respectivo sea agregado.
- {03} Inicializar el circuito como :

$$nodo(i) = i \qquad con\ i = 1,2,3,\ldots, n$$

  donde *n* es el número total de nodos, vértices o de ciudades. El nodo *(i)* se deja en el registro Best_S_so_Far.
- {04} El registro *S* contendrá el circuito que se va formando en la recursividad, debe ser inicializado en (1, [1], 0) que significa que es el primer nodo, el conjunto que contiene la ruta que se va formando contiene al nodo No. 1 cuyo peso es 0.
- {05} Llamar al procedimiento de Búsqueda_Exhaustiva_Rec para la búsqueda del costo mínimo y la ruta respectiva.
- {09} y {10} Recibir la información que llega a las variables del procedimiento
- {11} Si $k = n$ significa que ya se terminó de generar un circuito parcial, es decir deja de ser circuito parcial para ser circuito total y debe,

- {12} terminar de calcular el peso New_peso, al que se le sumará el peso que se encuentra en la posición A(S,nodo(k),1). Donde k es el último nodo del circuito. La columna debe ser la 1 para cerrar el circuito al nodo 1, porque éste fue el inicial.

- {14} a {17} Guardar en el registro $New\_S \leftarrow S$ siempre y cuando el peso *New_peso* recién calculado sea menor que el peso peso *B* anexando los datos recién calculados.

- {19} a {24} Se crean los circuitos con sus respectivos pesos y repetir la operación en 44 forma recursiva.

2.4.1 Método Ramificación y Acotamiento Ingenuo

El método Ramificación y Acotamiento Ingenuo es también llamado Naive Branch and Bound. Este se obtiene al hacerle una mejora al método de Búsqueda Exhaustiva Ingenua, ésta consiste en hacer una poda al árbol de búsqueda. ¿Pero cómo? Se observa cada una de las rutas parciales y si alguna de ellas ya es mayor que la de la mejor solución encontrada con anterioridad no hay razón para seguir buscando por ese camino. Por lo tanto se evita seguir en él y esto es lo que se llama poda del árbol de búsqueda. Esto se ve reflejado en el procedure de Ramificación_Acotamiento_Ingenuo_Rec que se muestra en la Figura 2.10

Como se puede ver el algoritmo es similar al de Búsqueda Exhaustiva Ingenua, por lo que solo se hará hincapié en las líneas que marcan la diferencia, que son:

- {14} a {18} Se compara si el peso recién calculado new_peso es menor que el peso B anterior. Si es menor guarda como Best_S_so_Far al calculado, pero si resulta ser mayor el peso nuevo New_peso entonces ya no llama a la recursividad. ¡He aquí el ahorro!, dado que ya no termina los circuitos parciales que están en estas circunstancias reduciendo las búsquedas de los $(n\text{-}1)!$ posibles caminos del método anterior. Pero cabe aclarar que también en el peor de los casos hace la búsqueda de los $(n\text{-}1)!$ caminos, siendo por ello de $O(n!)$.[Castañeda, 2000]

```
{01}Procedure Ramificación_Acotamiento_ingenuo_Rec (S, Best_S_so_far )
{02} Let S = ( k, [ i1, i2, ....., ik ], peso )
{03} Let Best_S_so_far = ( n, [ i1B, i2B, ....., inB ], pesoB )
{04} If k = n
{05} then
{06} new_peso = peso + A(ik, i1)
{07} if new_peso < pesoB
{08} then
{09} Best_S_so_far = ( k, [ i1, i2, ....., ik ], new_peso )
{10} end if
{11} else
{12} for all j not in [ i1, i2, ....., ik ]
{13} new_peso = peso + A(ik, j)
{14} if new_peso < pesoB
{15} then
{16} New_S = ( k + 1, [ i1, i2, ....., ik, j ], new_peso )
{17} Naive_Branch_and_Bound_Search ( New_S, Best_S_so_far )
{18} end if
{19} end for
{20} end if
{21} return
{22} end de Ramificación_Acotamiento_Ingenuo_Rec
```

Figura 2.10 Algoritmo recursivo de Búsqueda Exhaustiva Ingenua

2.4.2 Método Una Mejor Ramificación y Acotamiento

El método Una Mejor Ramificación y Acotamiento también recibe el nombre de A Better Branch and Bound. Para que sea aplicable, el costo debe estar bien definido, es decir, la matriz de adyacencia debe representar un grafo completo.

Este método está basado, como los anteriores, en un árbol de búsquedas, donde en cada etapa todas las posibles soluciones del problema son particionadas en dos subconjuntos, cada una representada por nodos en un árbol de decisiones. Al particionar en dos subconjuntos en cada etapa, el subconjunto de la izquierda contendrá una arista específica (i, j) y el subconjunto de la derecha no tendrá esa arista (i, j) . Esta ramificación se realiza de acuerdo con la siguiente heurística, la cual reduce la cantidad de búsquedas que conducen a la solución óptima. Después de ramificar se calculan las cotas para cada uno de los dos subconjuntos. En el próximo espacio de soluciones que se busque se debe escoger uno de ellos, que es el mínimo de las dos cotas a ser comparadas.

Este proceso se repite recursivamente hasta encontrar el ciclo hamiltoniano. Entonces, solamente dentro de los subconjuntos de soluciones se buscará la cota que sea más baja que el valor de la cota inicial o de arranque.

2.4.3 Heurística de Reducción para el método Una Mejor Ramificación y Acotamiento

Un ciclo hamiltoniano de longitud n contiene exactamente un elemento de cada fila de la matriz de costo o adyacencia W y exactamente un elemento de cada columna de W . Si a la

matriz $W(i,j)$ se le resta una constante q de cualquier fila o de cualquier columna, el costo de todos los ciclos Hamiltonianos (circuitos) son reducidos por q. Más aún los costos relativos de los diferentes ciclos restantes también son reducidos por q, lo mismo que el circuito óptimo. Si tal substracción es hecha de las filas y las columnas, de tal manera que cada fila y cada columna contenga al menos un cero, pero además que no guarde elementos negativos, entonces la cantidad restada será la cota más baja en el costo de cualquier solución. Este proceso de restar constantes de las filas y las columnas es llamada reducción, ver la Tabla 2.14.

i/j	V_1	V_2	V_3	V_4	V_5	V_6
V_1	∞	3	93	13	33	9
V_2	4	∞	77	42	21	16
V_3	45	17	∞	36	16	28
V_4	39	90	80	∞	56	7
V_5	28	46	88	33	∞	25
V_6	3	88	18	46	92	∞

Tabla 2.14 Matriz de Adyacencia de 6 ciudades para el PAV

Donde las V_i son los vértices, nodos o ciudades del PAV y la matriz $W(i,j)$ debe contener infinitos (∞) para excluir estos nodos que se encuentran sobre la diagonal. Esto se debe a que la matriz W representa un grafo completo. Para reducir la matriz de adyacencia o de costos se localizan los menores de cada fila y se restan a la matriz de costos. Se localizan los menores de cada columna y se restan a la matriz de costos, dejando la matriz reducida. Por otro lado se suman todos los menores de cada fila a todos los menores de cada columna para obtener la cota actual más baja, que es la cota más baja de todas las soluciones del problema. Para este ejemplo se restarían 3, 4, 16, 7, 25 y 3 de las filas y 15 y 8 de las columnas tres y cuatro, respectivamente, y la cota actual más baja es de 81. Ver las tablas 2.15 que muestra cada paso descrito anteriormente.

i/j	V_1	V_2	V_3	V_4	V_5	V_6	menores
V_1	∞	3	93	13	33	9	3
V_2	4	∞	77	42	21	16	4
V_3	45	17	∞	36	16	28	16
V_4	39	90	80	∞	56	7	7
V_5	28	46	88	33	∞	25	25
V_6	3	88	18	46	92	∞	3
							58

a) Matriz de adyacencia y menores de cada fila

i/j	V_1	V_2	V_3	V_4	V_5	V_6
V_1	∞	0	90	10	30	6
V_2	0	∞	73	38	17	12
V_3	29	1	∞	20	0	12
V_4	32	83	73	∞	49	0
V_5	3	21	63	8	∞	0
V_6	0	85	15	43	89	∞

b) Matriz de adyacencia después de restarle los menores de cada fila

i/j	V_1	V_2	V_3	V_4	V_5	V_6	menores
V_1	∞	0	90	10	30	6	3
V_2	0	∞	73	38	17	12	4
V_3	29	1	∞	20	0	12	16
V_4	32	83	73	∞	49	0	7
V_5	3	21	63	8	∞	0	25
V_6	0	85	15	43	89	∞	3
menores	0	0	15	8	0	0	81

c) Matriz de adyacencia y menores de cada columna

i/j	V_1	V_2	V_3	V_4	V_5	V_6
V_1	∞	0	75	2	30	6
V_2	0	∞	58	30	17	12
V_3	29	1	∞	12	0	12
V_4	32	83	58	∞	49	0
V_5	3	21	48	0	∞	0
V_6	0	85	0	35	89	∞

d) Matriz de adyacencia o de costos reducida después de restar menores de cada columna

Tablas 2.15 Pasos para calcular la primer Matriz Reducida

Ahora la pregunta sería: ¿Cómo se divide el conjunto de todas las soluciones en dos clases? Supóngase que se escogen los nodos $(6,3)$ entonces se divide el espacio de soluciones en dos, generando dos matrices. La matriz derecha será el conjunto que contendrá todas las soluciones que excluyen los nodos $(6,3)$, y por lo tanto, sabiendo que $(6,3)$ son excluidos, se debe cambiar la entrada de la matriz de costos en esa dirección a $W_{6,3} \leftarrow \infty$.

Lo que implica que a la matriz de costos se le restará 48 de la tercera columna y nada de la sexta fila, así se obtiene la cota actual más baja de 81 + 48 = 129 para todas las soluciones que excluyen los nodos $(6,3)$. En este mismo paso de división de soluciones se obtiene otra matriz, la matriz izquierda que contiene todas las soluciones que incluyen los nodos $(6,3)$, por lo que la sexta fila y la tercera columna debe ser borrada de la matriz de costos reducida, porque hora ya nunca se podrá ir desde el nodo 6 a ningún otro nodo o llegar al nodo 3 desde ningún otro nodo.

El resultado es una matriz de costos de dimensión (5x5) que es menor en uno que la anterior de (6x6), más aún, ya que todas las soluciones de este subconjunto usan los nodos (6, 3), los nodos (3, 6) no serán usados nunca más, por lo que se debe cambiar la entrada de la matriz de costos en esa dirección a $W_{3,6} \leftarrow \infty$ para prohibir estos nodos. El árbol de búsqueda binario en este momento es como se presenta en la Figura 2.11 con sus respectivas matrices derecha e izquierda.

i/j	V	V_2	V_4	V_5	V_6
V	∞	0	2	30	6
V_2	0	∞	30	17	12
V_3	29	1	12	0	12
V_4	32	83	∞	49	0
V_5	3	21	0	∞	0

a) Matriz izquierda

i/j	V	V_2	V_3	V_4	V_5	V_6
V	∞	0	75	2	30	6
V_2	0	∞	58	30	17	12
V_3	29	1	∞	12	0	12
V_4	32	83	58	∞	49	0
V_5	3	21	48	0	∞	0
V_6	0	85	0	35	89	

b) Matriz derecha

Figura 2.11 Primera división del espacio de búsqueda

Los nodos (6, 3) fueron usados para la solución porque, de todos los nodos, éstos causaron el mayor incremento en la cota actual más baja del subárbol derecho (en el árbol de búsqueda de la figura 2.8. De manera que la solución óptima se encontrará en las ramas izquierdas más que en las ramas derechas, las ramas izquierdas reducen la dimensión del problema, sin embargo, las ramas derechas sólo agregan otro (∞) y quizás otros pocos ceros sin cambiar la dimensión de la matriz. Para seleccionar los mejores nodos, se escoge el cero que cuando cambie a (∞) permita restarle a esta fila y esa columna la mayor cantidad posible, esto se muestra en las Tablas 2.16

Así, en la matriz de costos (5x5) que representa el problema del subárbol de la matriz izquierda de la Figura 2.11, el mejor cero está en (4, 6) y éste es cambiado a (∞). Esto implica que a la matriz de costos se le resta 32 de la cuarta fila y nada de la sexta columna. Este cero es el que tiene la mayor suma de los menores de entre todos los ceros que hay en la matriz. Por lo tanto la siguiente división se hará con los nodos (4, 6). Esto incrementa la cota actual más baja de todas las soluciones que incluyen al nodo (6, 3) y que excluyen los nodos (4, 6) a 81 + 32 = 113 .Esto decrementa la dimensión de la matriz en la rama izquierda a una matriz de (4x4), ya que debe borrarse la fila 4 y la columna 6. Esta situación se presenta en las Tablas 2.17, donde se muestra el árbol generado hasta el momento con sus respectivas matrices, derecha e izquierda.

i/j	V	V_2	V_3	V_4	V_5	V_6	menores
V	∞	0	75	2	30	6	2
V_2	0	∞	58	30	17	12	12
V_3	29	1	∞	12	0	12	1
V_4	32	83	58	∞	49	0	32
V_5	3	21	48	0	∞	0	0
V_6	0	85	0	35	89	∞	0
menores	0	1	48	2	17	0	

a) Matriz de costos reducida

Coordenadas	Cota mejornd	Coordendas	Cota mejornd
1,2	2+1=3		
2,1	12+0=12		
3,5	1+17=18		
4,6	32+0=32		
5,4	0+2=2	5,6	0+0
6,1	0+0=0	6,3	0+48

b) Coordenadas del mejor y su respectiva cota

Tablas 2.16 Selecciona el primer mejor cero

i/j	V	V_2	V_4	V_5
V	∞	0	2	30
V_2	0	∞	30	17
V_3	29	1	12	0
V_5	3	21	0	

a) Matriz izquierda

i/j	V	V_2	V_4	V_5	V_6
V	∞	0	2	30	6
V_2	0	∞	30	17	12
V_3	29	1	12	0	12
V_4	32	83	∞	49	0
V_5	3	21	0	∞	0

b) Matriz derecha

Tablas 2.17 Segunda división del espacio de búsqueda

Note que ya que los nodos (4, 6) y (6, 3) son incluidos en la solución, los nodos (3, 4) no se usarán nunca más, esto se refuerza poniendo como entrada en $W_{3,4} \leftarrow \infty$ .En general, si el nodo que se agrega a la ruta parcial va desde i_u a j_1 y la ruta parcial contiene caminos $(i_1, i_{2,\ldots}, i_n)$ y $(j_1, j_{2,\ldots}, j_k)$, los nodos cuyo uso deben prohibirse están en (j_k, i_1).

Cuando se hace una nueva división se encuentran nuevamente los mejores nodos que 49 son (2,1), así que a la matriz de costos derecha se le prohíben los nodos (2, 1) al hacer $W_{2,1} \leftarrow \infty$ y esto hace que se le reste 17 a la segunda fila y 3 a la primera columna como se muestra en las Tablas 2.18

Solución con (6,3) y (4,6)

i/j	V	V_2	V_4	V_5	Coordenadas	Cota major nd
V	∞	0	2	30	1,2	2+1=3
V_2	0	∞	30	17	2,1	17+3=20
V_3	29	1	12	0	3,5	1+17=18
V_5	3	21	0	∞	5,4	3+2=5

a) (r,c) =(2,1) cota mejorad=20

Solución con (6,3) y (4,6) y (2,1)

i/j	V_2	V_4	V_5
V	0	2	30
V_3	1	12	0
V_5	21	0	∞

Solución con (6,3),(4,6) y sin (2,1)

i/j	V	V_2	V_4	V_5
V	∞	0	2	30
V_2	0	∞	13	0
V_3	26	1	12	0
V_5	0	21	0	∞

Tablas2.18 Selección del tercer mejor cero y división del espacio de búsqueda

Después de dividir el espacio de búsqueda en la matriz derecha e izquierda de la Tabla 2.18, la matriz de costos izquierda es una matriz de (3x3). En la que se han incluido los nodos (2,1) a la solución izquierda por lo que los nodos (1, 2) se prohíben en la matriz derecha al hacer $W_{1,2} \leftarrow \infty$

Se realiza nuevamente la reducción de la matriz de (3x3) restando 1 de la columna dos y 2 de la fila uno ahora quedando una nueva matriz de costos reducida. La cota actual más baja también se actualiza, dado que en la solución de este subconjunto es de 81 + 1+2 = 84, como se muestra en las Tablas 2.19.

Solución con (6,3) y (4,6) y (2,1)

i/j	V_2	V_4	V_5	menores
V	0	2	30	2
V_3	1	12	0	0
V_5	21	0	∞	0

a)Matriz de adyacencia y menores de cada fila

Solución con (6,3),(4,6) y sin (2,1)

i/j	V_2	V_4	V_5
V	0	2	30
V_3	1	12	0
V_5	21	0	

b)Matriz después de restar los menores de fila

i/j	V_2	V_4	V_5	menores
V	0	2	30	2
V_3	1	1 2	0	0
V_5	21	0	∞	0
menores	1	0	0	3

c)Matriz de adyacencia y menores de cada columna

Solución con (6,3), (4,6) y (2,1)

i/j	V_2	V_4	V_5
V	∞	0	28
V_3	1	∞	0
V_5	21	0	∞

d) Matriz después de restar menores de columna

Tablas 2.19 Pasos para calcular la tercer matriz reducida

Nótese que después de $(n-2)$ nodos seleccionados, la matriz de costos es una matriz de dimensión (2x2), en este momento las dos últimas parejas de nodos se forzan para formar un camino. En el ejemplo se tienen los nodos (6, 3), (4, 6), (2, 1), y (1, 4) así que solo queda agregar los nodos (3,5) y (5,2) para completar la ruta del PAV.

Se obtiene una ruta al ir guardando las filas que se van quitando, es decir cuando se encontraron los nodos (6,3) se guardo en la dirección de la fila 6 la dirección 3, cuando se encontraron los nodos (4,6) se guardo en la dirección de la fila 4 dirección 6 y así sucesivamente, como se muestra en la Tabla 2.20.

filas	1	2	3	4	5	6
quitarfilas						3

a) El nodo 6 va al nodo 3

filas	1	2	3	4	5	6
quitarfilas		1		6		3

b) El nodo 4 va al nodo 6

filas	1	2	3	4	5	6
quitarfilas		1		6		3

c) El nodo 2 va al nodo 1

filas	1	2	3	4	5	6
quitarfilas	4	1		6		3

d) El nodo 1 va al nodo 4

filas	1	2	3	4	5	6
quitarfilas	4	1	5	6		3

e) El nodo 3 va al nodo 5

filas	1	2	3	4	5	6
quitarfilas	4	1	5	6	2	3

f) El nodo 5 va al nodo 2

Tabla 2.20 Ruta encontrada a partir de los mejores nodos seleccionados

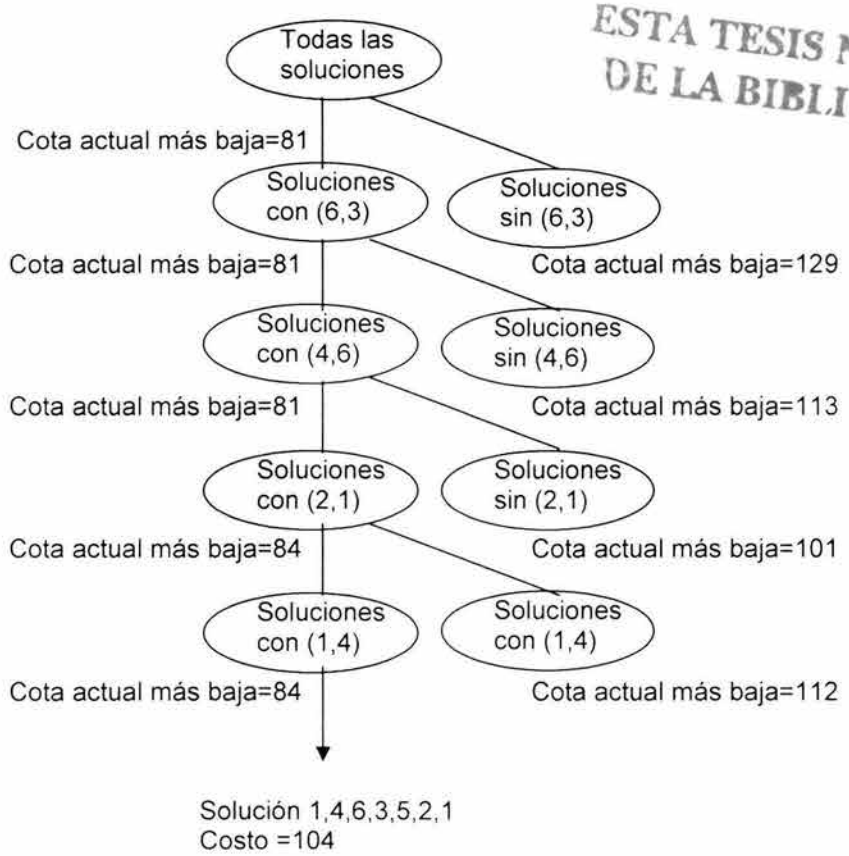


Figura 2.12 Árbol binario de A Better Branch and Bound

Con la historia de las filas que se quitaron se encuentra la ruta final, siguiendo las direcciones. La ruta final para este ejemplo es 1, 4, 6, 3, 5, 2, 1, cuyo costo es de 104, los nodos en el árbol de búsqueda se muestran en la figura 2.9 [Castañeda, 2000]. Al analizar el costo y las cotas actuales más bajas del árbol binario de la figura 2.9 puede notarse que hay

un subárbol con la cota actual más baja = 101 que es menor que el costo de la ruta que se obtuvo por lo que ese subárbol debe ser examinado y expandido porque existe la posibilidad de que haya otra ruta con un costo menor que el de la ruta anterior.

La cota actual más baja = 101 incluye los nodos (6, 3) y (4,6) pero excluye los nodos (2, 1), la matriz de costos asociada a estos nodos es la que se presenta en la Tabla 2.21.

Solución con (6,3), (4,6) y sin (2,1)

i/j	V	V_2	V_4	V_5
V	∞	0	2	30
V_2	∞	∞	13	0
V_3	26	1	∞	0
V_5	0	21	0	∞

b) Matriz derecha

Tabla 2.21 Matriz de costos del subárbol con cota actual más baja = 101

Esta matriz de costos (4x4) debe ser dividida al encontrar el mejor cero en (5, 1), esto excluye los nodos (5, 1) y suma 26 a la cota actual mas baja dando 101+ 26 = 127 generando el subárbol que se muestra en la Figura 2.13.

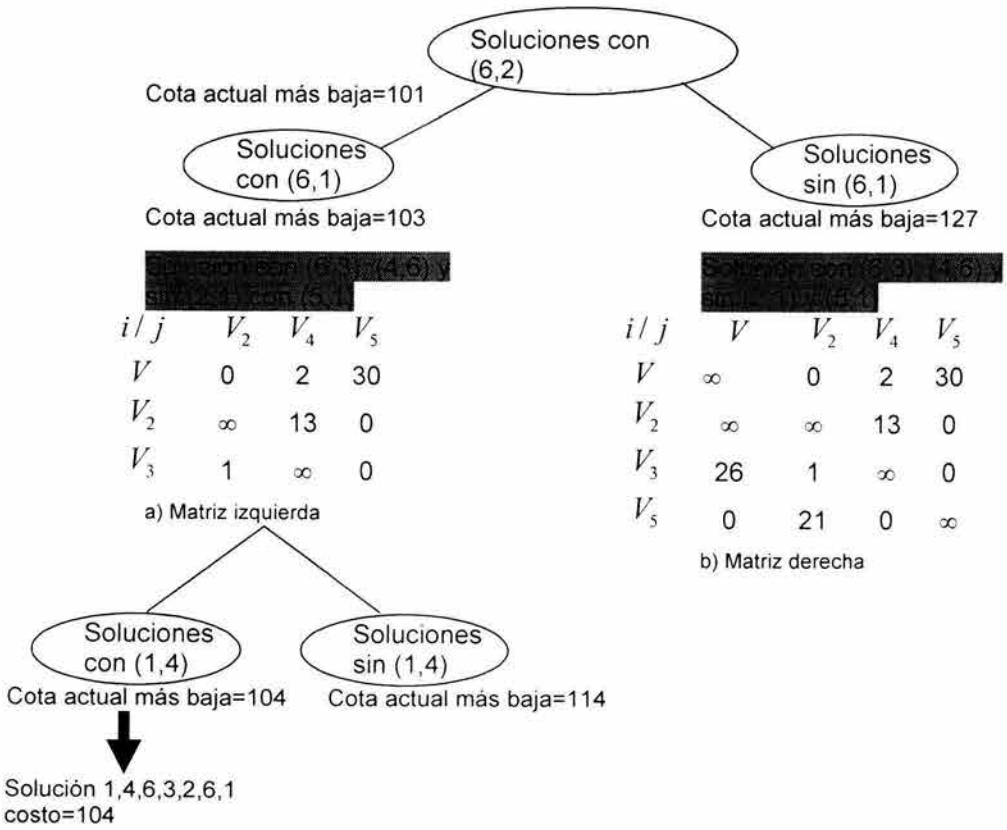


Figura 2.13 División del espacio de búsqueda del subárbol con cota actual mas baja = 101

De esta manera se obtienen dos soluciones óptimas o dos rutas diferentes (1, 4, 6, 3, 5, 2, 1) y (1, 4, 6, 3, 2, 5, 1) cada una con un costo de 104, que es el costo menor de todas las rutas. Como puede verse en este ejemplo para la solución del PAV no necesariamente se obtiene una ruta como solución óptima, puede haber más de una solución óptima con un solo costo mínimo y rutas diferentes. También cabe aclarar que no siempre la solución se encuentra en la primera solución como ocurrió en este ejemplo, en este caso particular se examinaron 13 nodos, lo cual es un gran ahorro de búsqueda.

Se mencionó anteriormente que la matriz después de reducir su dimensión una y otra vez llegaba a una matriz de (2x2). Entonces se forzaban los últimos pares de nodos para cerrar la ruta correspondiente, dado que ninguna otra división del espacio de búsqueda podía realizarse. Así que, cuando esto ocurre, simplemente se agregan los últimos nodos sin que se puedan seleccionar y se forma el ciclo Hamiltoniano. Esto se debe a que la matriz de (2x2) sólo puede tener dos formas como se ve en la Tabla 2.22.

	w	x
u	∞	0
v	0	∞

a)

	w	x
u	0	∞
v	∞	0

b)

Tabla 2.22

En esta matriz de costos (2x2) los nodos u, v, w, y x generan dos casos, en cualquiera de los estos solamente se requiere una entrada para cerrar el ciclo Hamiltoniano. Véase la fila y columna (1,1) para determinar cuáles nodos serán incluidos. Así, si $W(1,1) = \infty$ agregar los nodos (u, x) y (v, w) que son los nodos que tienen cero, en caso contrario agregar los nodos (u, w) y (v, x) para cerrar la ruta que forma el ciclo Hamiltoniano .

2.9.2 Algoritmo del método Una Mejor Ramificación y Acotamiento

Este algoritmo puede dividirse en varios subalgoritmos que resuelvan tres preguntas importantes porque son ellas las que determinan la heurística a seguir:

a) ¿Cómo acotar los pesos de la solución de cada subárbol?

b) ¿Cómo se representa el conjunto de soluciones?

c) ¿Cómo se divide y cómo se escogen los mejores nodos en el espacio de búsqueda?

Estas preguntas se irán contestando conforme se describan las etapas a seguir. Primero se acotan las soluciones, es decir, se reduce de la matriz W restando una constante de manera que los valores remanentes no sean negativos. Esto cambiará los pesos de cada viaje, pero no el conjunto de viajes legales y sus pesos relativos. Se describe en la Figura 2.14 la función Reduce que contesta el inciso a.

```
{01} Function Reduce (W)
{02}begin
{03} sumamenores = 0;
{04} for i ← 1 to ntamanio do
{05} begin
{06} filred(i) = menor de los elementos de la fila i
{07} if (filred(i) > 0 )
{08} begin
{09} resta filred(i) de cada elemento de la matriz A en la fila i
{10} sumamenores = sumamenores + filred(i);
{11} end del if
{12} end del for
{13} for j ← 1 to ntamanio do
{14} begin
{15} colred(i) = menor de los elementos de la columna j
{16} if (colred(i) > 0 )
{17} begin
{18} resta colred(j) de cada elemento de la matriz W en la columna j
{19} sumamenores = sumamenores + colred(j);
{20} end del if
{21} end del for
{22} return sumamenores;
{23}end de la function Reduce
```

Figura 2.14 Algoritmo Reduce

Se describirán las líneas más importantes

-{01} En sumamenores se guardará el valor de la reducción
-{04} ntamanio es el tamaño de la matriz de adyacencia W
-{04} a -{12} Se localiza el menor de cada fila i guardándolo en filred(i), y se resta este menor de cada elemento de la matriz A en la fila i, tantas veces como filas existan. Acumulando en sumamenores todos los menores de todas las filas.
-{13} a – {21}Se realiza la misma operación con las columnas guardando los menores de cada columna en colred(i), acumulando en sumamenores todos los menores de todas las columnas.

La matriz de adyacencia o de costos es usada para representar el conjunto de soluciones, lo que responde el inciso b. Para ramificar o dividir el espacio de soluciones se debe determinar el mejor cero, es decir, en que posición (r,c) se encuentra el cero que es capaz de reducir la matriz W cuando se coloca un ∞ de manera que maximice la cantidad a ser restada desde la fila y la columna. La posición del mejor cero determina cuáles son los mejores nodos . Esto responde la mitad del inciso c y se lleva a cabo con el siguiente procedimiento de la Figura 2.15.

```
{01} Procedure selecciona_los_mejores_nodos ( ntamanio, r, c, a, cotamejornd)
{02} begin
{03} cotamejornd = -∞;
{04} for i ← 1 to ntamanio do // fila
{05} for j ← 1 to ntamanio do // columna
{06} if A (i,j) = 0
{07} begin
{08} buscar menor_en_fil;
{09} buscar menor_en_col;
{10} total = menor_en_fil + menor_en_col;
{11} if total > cotamejornd
{12} begin
{13} cotamejornd = menor_en_fil + menor_en_col;
{14} r ← i;
{15} c ← j;
{16} end del if
{17} end del if
{18} A(i,j) - menor_en_ fil;
{19} A(i,j) - menor_en_col;
{20} end del Procedure selecciona_ mejor_cero
```

Figura 2.15 Algoritmo Selecciona los mejores nodos

Se hará una breve descripción de las líneas más importantes [Castañeda, 2000]:

- {04} a {09} Para cada fila y para cada columna se localiza el mejor cero. Se localiza un cero en $A(i,j)$ en la fila i, en menor_en_fil se guarda el menor de la fila exceptuando $A(i,j)$ que tiene 0. Es decir, se localiza el menor de cada fila diferente de cero. Si hay dos o más ceros el menor es cero. En la columna j donde se localizó el cero, en menor_en_col se guarda el menor de la columna, exceptuando $A(i,j)$ que tiene 0. Es decir, se localiza el menor de cada columna diferente de cero.

Si hay dos o más ceros el menor es cero.

- {10} a {17} Si total que es la suma de menor_en_fil y menor_en_col resulta ser mayor que la cotamejornd, se toma esta suma como la nueva cotamejornd. Guardando además en r el índice de la fila que será uno de los mejores nodos y en c al índice de la columna que será el otro mejor nodo

- {18} Resta a la matriz de costos reducida $A(i,j)$ menor_en_fil de cada elemento de la fila *r*, excepto a ceros e infinitos.

- {19} Resta a la matriz de costos reducida $A(i,j)$ menor_en_col de cada elemento de la columna *c*, excepto a ceros e infinitos.

Tanto la función Reduce como el procedimiento selecciona_ mejor_cero son llamados en el procedimiento recursivo llamado ABetterBranchBoundRec, que contesta la otra mitad del inciso c, éste se describe en la Figura 2.16.

```
{01} Procedure ABetterBranchBoundRec (numnds, W, new_reg)
{02} begin
{03} cotaactual = cotaactual + reduce (A);
{04} if cotaactual < cotainf
{05} if numnds = n_vertices - 2
{06} begin
{07} Los dos últimos nodos son forzados;
{08} Se guarda la nueva solución;
{09} cotainf ← cotaactual;
{09} end
{10} else
{11} begin
{12} selecciona_mejor_cero(ntamanio, r, c, a, cotamejornd);
{14} Se prevén y prohiben subciclos
{15} NewA ← A- columna c - fila r;
{16} BetterBranchBoundRec(numnds+1, New_A, new_reg);
{17} restaura A agregando columna c y fila r;
{18} if cotamasbaja < cotainf
{19} begin
{20} A(r,c) ← INFINITO;
{21} BetterBranchBoundRec(numnds,a,new_reg );
{22} a(r,c) ← 0;
{23} end del if cotamasbaja < cotainf;
{24} end del if
{25} end del if numnds = n_vertices - 2
{26} restaura_matriz_izq de la reducción;
{27} end del Procedure BetterBranchBoundRec
```

Figura 2.16 Algoritmo ABetterBranchBoundRec

Se hará una breve descripción de las líneas más importantes:

- {15} En NewA se almacena la matriz *A* después de borrar la fila r y la columna c.

- 16} Se genera el subárbol izquierdo al hacer el llamado recursivo al procedimiento BetterBranchBoundRec; con el número de nodos más uno, y la matriz New_A.

- {21} Aquí la segunda recursividad manda a la matriz derecha "a" con el mismo número

de nodos.

-{22} Restaura en la matriz "a" los nodos excluidos que fueron en su oportunidad la dirección del mejor cero, haciendo a la matriz en la fila r y la columna c igual a cero.

El método Una Mejor Ramificación y Acotamiento es, en el peor de los casos, básicamente el método de Búsqueda Exhaustiva, así que en ese caso se examinaría todo el árbol para analizar todas las posibles soluciones. Para un problema PAV Asimétrico de n ciudades hay $(n-1)!$ ciclos Hamiltonianos diferentes; y en el peor de los casos el algoritmo de Una Mejor Ramificación y Acotamiento es de $O(n!)$. En un caso típico, sin embargo, la situación no es

tan mala. Por ejemplo, para el caso particular de 6 ciudades estudiado anteriormente se examinaron solo 13 nodos de los 120 ciclos distintos para este problema de 6 ciudades $[(n\text{-}1!) = (6\text{-}1)! = 120]$. Por lo que fueron 13 reducciones, 13 divisiones del espacio de búsqueda, etc.que es mucho menor que 120 = 5! [Castañeda, 2000]. De hecho el tiempo de ejecución es extremadamente dependiente de las instancias del problema PAV [Castañeda, 2000]. Se sugiere ver el Capítulo 1, donde se puede apreciar como crece rápidamente el tiempo de ejecución con el tamaño de la matriz de adyacencia (número de ciudades).

2.5 Programación Dinámica

La programación dinámica, desarrollada por Richard E. Bellman en la década de los 50, está considerada como una potente herramienta de optimización de carácter muy general. Constituye una disciplina muy importante de las matemáticas aplicadas y la investigación de operaciones, y su método estándar se aplica en áreas tan dispares como ingeniería, inteligencia artificial, economía, gestión, etc.

El precio que hay que pagar por tanta flexibilidad es el alto costo computacional del método: el tiempo de cálculo y los recursos de memoria empleados son muy grandes y crecen exponencialmente con el tamaño del problema. Aunque para algunas aplicaciones la programación dinámica se puede aplicar analíticamente, en general la solución se debe de encontrar numéricamente, teniendo presente el problema anteriormente comentado. El tiempo computacional que se consigue con los procesadores convencionales sigue siendo demasiado alto para la mayoría de los casos de interés práctico.

Sin embargo, en los últimos años el abaratamiento de los costos en las computadoras y el avance por todos conocido en la tecnología ha permitido considerar la posibilidad de realizar procesamiento paralelo de una forma sencilla y general. Por tanto, una computadora con procesamiento paralelo puede reducir en gran medida el tiempo computacional, cuando se resuelven problemas de programación dinámica de gran escala. Esto es debido a que hay un gran número de operaciones que se realizan en paralelo durante la evaluación de la fórmula recursiva de la programación dinámica.

La teoría computacional de la programación dinámica desde el punto de vista de la computación paralela fue examinada por Larson en 1973, aunque los algoritmos que propuso únicamente son aplicables a arquitecturas paralelas muy poco flexibles y muy caras de implementar. Desde comienzos de la década de los 90 ha existido una tendencia creciente a alejarse de las supercomputadoras especializadas paralelos (supercomputadores vectoriales y procesadores masivamente paralelos, MPPs), debido a sus elevados costos en hardware, mantenimiento y programación. Una alternativa de menor costo ampliamente utilizada y consolidada son los clusters de estaciones de trabajo . Entre los motivos que han hecho posible esta transición cabe destacar el gran progreso en la disponibilidad de componentes de un alto rendimiento para estaciones de trabajo y redes de interconexión. Gracias a estos avances, se ha logrado que un cluster sea hoy día un sistema muy atractivo en cuanto a su relación costo/rendimiento para el procesamiento en paralelo. Utilizan un software estándar y portable, como UNIX (Linux) y algún entorno de programación paralelo, como pueden ser PVM (Parallel Virtual Machine) o MPI (Message Passing Interface) para implementar el paso de mensajes y la sincronización entre procesos. En este entorno, la eficiencia de las aplicaciones paralelas es mayor cuando la carga computacional se distribuye equitativamente entre las estaciones de trabajo que constituyen el cluster y se

minimiza el número de comunicaciones. Por tanto, el sistema de interconexión que soporta el intercambio de mensajes debe ser lo suficientemente rápido para evitar ser el cuello de botella de la aplicación paralela.

2.5.1 Control óptimo y programación dinámica

El objetivo de la programación dinámica es resolver una familia de problemas de optimización con restricciones conocidas como problemas de decisión secuencial o multietapa . La principal característica de estos procedimientos es que cualquier decisión realizada en un instante de tiempo se ve afectada por las decisiones predecesoras y afecta a sus sucesoras. La programación dinámica se basa en el Principio de Óptimo de Bellman [Bellman,1957], tal como se ilustra en la Figura 2.17

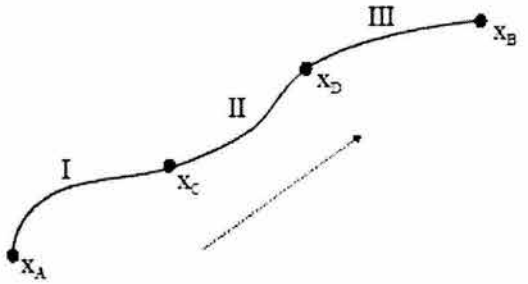


Fig. 2.17 Principio de Óptimo de Bellman: si I-II-III es la trayectoria óptima desde el estado x_A hasta el estado x_B, de acuerdo con una función de costo dada, entonces II es la trayectoria óptima para el subproblema $x_C - x_D$.

El método para dar solución a un problema consiste en ir resolviendo recursivamente subproblemas cada vez mayores hasta que se soluciona el problema completamente.

2.5.2 Planteamiento de la solución del PAV como un problema de programación dinámica.

A continuación se plantea la solución del problema del agente viajero mediante programación dinámica.

Sea $N_j = (2,3,\ldots, j-1, j+1,\ldots,N)$ y sea S un subconjunto de N_j que contiene i miembros.

Se define el valor óptimo de la función $f_i(j,S)$ como:

$f_i(j,S)$ = La longitud de la trayectoria mas corta de la ciudad 1 a la ciudad j a traves de un conjunto S de ciudades intermedias (la etapa i representa el numero de ciudades i en S) (1)

La formulación de las ecuaciones recursivas es la siguiente:

$$f_i(j,S) = \min_{k \in S} \{ f_{i-1}(k, S - \{k\} + d_{kj} \} \qquad i = 1,2,\ldots,N-2;\ j \neq 1;\ S \subseteq N \qquad (2)$$

Condiciones de frontera:

$$f_0(j,-) = d_{1j} \tag{3}$$

La trayectoria más corta va a estar dada por:

$$\min_{j=2,3,\ldots,N}\left\{f_{N-2}\left(j,N_j\right) + d_{j1}\right\} \tag{4}$$

Podemos justificar intituivamente la relación de recurrencia. Supongamos que queremos calcular $f_i(j,S)$. Consideramos las trayectorias más cortas de la ciudad 1 a la ciudad j vía S que tiene a la ciudad k como predecesora inmediata de la ciudad j. Ya que las ciudades en $S-\{k\}$ deben visitarse en un orden óptimo, la longitud de esta trayectoria es $f_{i-1}(k,S-\{k\}) + d_{kj}$. Más aún, ya que somos libres de seleccionar la ciudad k óptimamente, es claro que $f_i(j,S)$ está definido correctamente.

Entonces, para calcular la longitud de la trayectoria más corta, comenzamos por calcular $f_1(j,S)$ (para todo par j , S) a partir de los valores de f_0. Entonces calculamos $f_2(j,S)$ (para todo par j , S) a partir de los valores de f_i. Continuamos de la misma manera hasta $f_{N-2}(j,N_j)$ para cada ciudad j.

La longitud de la trayectoria más corta está dada por (4). Ya que alguna j debe ser la última visitada antes de regresar a la ciudad 1. El recorrido de trayectoria más corta se puede obtener guardando en cada etapa y cada estado la ciudad k que minimiza la ecuación recursiva.

Esto lo ilustramos con el siguiente ejemplo.

A continuación se da la matriz de distancias de la red que consiste de 5 ciudades:

	1	2	3	4	5
1	0	3	1	5	4
2	1	0	5	4	3
3	5	4	0	2	1
4	3	1	3	0	3
5	5	2	4	1	0

Tabla 2.23

Los cálculos son los siguientes, la política óptima se da entre paréntesis:

Condiciones de frontera:

$$f_0(2,-) = d_{12} = 3(1)$$
$$f_0(3,-) = d_{13} = 1(1)$$
$$f_0(4,-) = d_{14} = 5(1)$$
$$f_0(5,-) = d_{12} = 4(1)$$

Etapa 1

$f_1(2,\{3\}) = f_0(3,-) + d_{32}$=1+4=5(3)

$f_1(2,\{4\}) = f_0(4,-) + d_{42} = 5+1 = 6(4)$

$f_1(2,\{5\}) = f_0(5,-) + d_{52} = 4+2 = 6(4)$

$f_1(3,\{2\}) = f_0(3,-) + d_{23} = 3+5 = 8(2)$

$f_1(3,\{4\}) = f_0(4,-) + d_{43} = 5+3 = 8(4)$

$f_1(3,\{5\}) = f_0(5,-) + d_{53} = 4+4 = 8(5)$

$f_1(4,\{2\}) = f_0(2,-) + d_{24} = 3+4 = 7(2)$

$f_1(4,\{3\}) = f_0(3,-) + d_{34} = 1+2 = 3(3)$

$f_1(4,\{5\}) = f_0(5,-) + d_{54} = 4+1 = 5(5)$

$f_1(5,\{2\}) = f_0(2,-) + d_{25} = 3+3 = 6(2)$

$f_1(5,\{3\}) = f_0(3,-) + d_{35} = 1+1 = 2(3)$

$f_1(5,\{4\}) = f_0(4,-) + d_{45} = 5+3 = 8(4)$

Etapa 2

$f_2(2,\{3,4\}) = \min\{f_1(3,\{4\}) + d_{32}, f_1(4,\{3\}) + d_{42}\} = \min\{8+4, 3+1\} = 4(4)$

$f_2(2,\{3,5\}) = \min\{f_1(3,\{5\}) + d_{32}, f_1(5,\{3\}) + d_{52}\} = \min\{8+4, 2+2\} = 4(5)$

$f_2(2,\{4,5\}) = \min\{f_1(4,\{5\}) + d_{42}, f_1(5,\{4\}) + d_{52}\} = \min\{5+1, 8+2\} = 6(4)$

$f_2(3,\{2,4\}) = \min\{f_1(2,\{4\}) + d_{23}, f_1(4,\{2\}) + d_{43}\} = \min\{6+5, 7+3\} = 10(4)$

$f_2(3,\{2,5\}) = \min\{f_1(2,\{5\}) + d_{23}, f_1(5,\{2\}) + d_{53}\} = \min\{6+5, 6+4\} = 10(5)$

$f_2(3,\{4,5\}) = \min\{f_1(4,\{5\}) + d_{43}, f_1(5,\{4\}) + d_{53}\} = \min\{5+3, 8+4\} = 8(4)$

$f_2(4,\{2,3\}) = \min\{f_1(2,\{3\}) + d_{24}, f_1(3,\{2\}) + d_{34}\} = \min\{5+4, 8+2\} = 9(2)$

$f_2(4,\{2,5\}) = \min\{f_1(2,\{5\}) + d_{24}, f_1(5,\{2\}) + d_{54}\} = \min\{6+4, 6+1\} = 7(5)$

$f_2(4,\{3,5\}) = \min\{f_1(3,\{5\}) + d_{34}, f_1(5,\{3\}) + d_{54}\} = \min\{8+2, 2+1\} = 3(5)$

$f_2(5,\{2,3\}) = \min\{f_1(2,\{3\}) + d_{25}, f_1(3,\{2\}) + d_{35}\} = \min\{5+3, 8+1\} = 8(2)$

$f_2(5,\{2,4\}) = \min\{f_1(2,\{4\}) + d_{25}, f_1(4,\{2\}) + d_{45}\} = \min\{6+3, 7+3\} = 9(2)$

$f_2(5,\{3,4\}) = \min\{f_1(3,\{4\}) + d_{35}, f_1(4,\{3\}) + d_{45}\} = \min\{8+1, 3+3\} = 6(4)$

Etapa 3

$$f_3(2,\{3,4,5\}) = \min\{f_2(3,\{4,5\}) + d_{32}, f_2(4,\{3,5\}) + d_{42}, f_2(5,\{3,4\}) + f_1(4,\{3\}) + d_{52}\}$$
$$= \min\{8+4, 3+1, 6+2\} = 4(4)$$
$$f_3(3,\{2,4,5\}) = \min\{6+5, 7+3, 9+4\} = 10(4)$$
$$f_3(4,\{2,3,5\}) = \min\{4+4, 10+2, 8+1\} = 8(2)$$
$$f_3(5,\{2,3,4\}) = \min\{4+3, 10+1, 9+3\} = 7(2)$$

Entonces se tiene que:

$$Min\{f_3(j,(2,3,4,5) - \{j\} + d_{j1}\} = \min\{4+1, 10+5, 8+3, 7+5\} = 5(2)$$

El circuito hamiltoniano de menor longitud es 5 con las ciudades:

1 ⟶ 3 ⟶ 5 ⟶ 4 ⟶ 2 ⟶ 1

Complejidad Computacional del algoritmo de programación dinámica

En la etapa i $f_i(j,S)$ se debe evaluar para $(N-1)(i^{N-2})$ para diferentes parejas j, S. Cada evaluación requiere i sumas e $i-1$ comparaciones. Entonces para todas las etapas tenemos el siguiente número de sumas y comparaciones requeridas:

Número de sumas:

$$= (N-1)\sum_{i=1}^{N-2}\binom{N-2}{i} = (N-1)(N-2)\sum_{i=1}^{N-2}\frac{(N-3)!}{(i-1)(N-2-i)!}$$
$$= (N-1)(N-2)\sum_{i=1}^{N-2}\binom{N-3}{i-1} = (N-1)(N-2)\sum_{j=0}^{N-3}\binom{N-3}{j} = (N-1)(N-2)2^{N-3}$$

Número de comparaciones:

$$= (N-1)\sum_{i=1}^{N-2}(i-1)\binom{N-2}{i} = numero\ \ de\ \ sumas - (N-1)\sum_{i=1}^{N-2}(i-1)\binom{N-2}{i} =$$
$$= (N-1)(N-2)2^{N-3} - (N-1)(2^{N-2}-1)$$
$$= (N-1)2^{N-3}\left[(N-2)-2\right] + (N-1) \approx (N-1)(N-4)2^{N-3}$$

No se consideró el número de sumas $(N-1)$ y comparaciones $(N-2)$ necesarias para calcular la respuesta con el circuito óptimo, pero no es necesario, pues es una cantidad pequeña que no afecta el resultado final. Si consideramos un problema de 20 ciudades se requiere aproximadamente 85 millones de operaciones.

En realidad el límite en el tamaño del problema tiene que ver más con la capacidad de almacenamiento que con la cantidad de operaciones. En la etapa i, se requieren almacenar

$(N-1)C_i^{N-2}$ lugares para almacenar f_i. Para *N* par el número de lugares requeridos es máximo cuando $i=\frac{(N-2)}{2}$. En el caso de $N=20$ se necesitan aproximadamente 925 mil lugares de almacenamiento para guardar f_i.

Un procedimiento de doblamiento en el caso de distancias simétricas.
En esta sección consideramos un procedimiento de doblamiento para el problema del agente viajero que es válido cuando se tiene una matriz de distancias simétricas, es decir $d_{ij}=d_{ji}$ para toda *i,j*.
Suponemos que N es par (se requieren modificaciones menores para N impar). Supongamos que $f_1, f_2, ..., f_{\frac{(N-2)}{2}}$ se han calculado para (2) y (3). Considere un recorrido que empieza y termina en la ciudad 1. Una ciudad digamos *j*, será visitada en algún punto medio del recorrido, entonces la longitud de los recorridos más cortos desde la ciudad 1 a la ciudad *j* a través de un conjunto de ciudades intermedias $\frac{(N-2)}{2}$ está dado por $f_{\frac{(N-2)}{2}}$. Entonces la longitud del recorrido más corto está dado por:

$$\min_{j=2,3,...N}\left\{\min_{S\subseteq N_j}\left\{f_{\frac{(N-2)}{2}}(j,S)+f_{\frac{(N-2)}{2}}(j,N_j-S)\right\}\right\}$$

Comparamos los requerimientos computacionales del procedimiento de doblamiento con los procedimientos normales. Se requiere el siguiente número de sumas y comparaciones para el procedimiento de doblamiento:

$$numero\ \ de\ \ sumas=(N-1)\sum_{i=1}^{N-2/2} i\binom{N-2}{i}+(N-1)\frac{1}{2}\binom{N-2}{(N-2)/2}$$

$$numero\ \ de\ \ comparaciones=(N-1)\sum_{i=1}^{N-2/2}(i-1)\binom{N-2}{i}+(N-1)\frac{1}{2}\binom{N-2}{(N-2)/2}-1$$

Donde $\binom{N-2}{(N-2)/2}/2$ es el número de particiones no ordenadas de un conjunto de $N-2$ objetos en dos conjuntos de $(N-2)/2$ objetos cada uno. En el caso de *N=20*, se necesitan un total de aproximadamente 43 millones de operaciones. De esta manera este procedimiento es considerablemente más eficiente, solamente se tiene que la cantidad de almacenamiento requerida es la misma que el otro procedimiento.

Capítulo 3. Técnicas Heurísticas para resolver el problema del agente viajero

En este capítulo, se describen las principales técnicas heurísticas de solución de problemas combinatorios entre los que figuran el problema del agente viajero. Comienzo por describir los fundamentos de los algoritmos heurísticos y enseguida realizo una clasificación de algunos de las técnicas heurísticas y metaheurísticos que son los más utilizados y reconocidos en la optimización combinatoria. Cabe mencionar que las técnicas heurísticas están exclusivamente diseñadas para un problema en específico con características muy especiales mientras que las técnicas metaheurísticas son técnicas que ayudan a resolver una clase de problemas y en este sentido estas técnicas también se tienen que adecuar al problema que se quiere resolver.

3.1 Introducción

Algunas clases de problemas de optimización son relativamente fáciles de resolver. Este es el caso, por ejemplo, de los *problemas lineales*, en los que tanto la función objetivo como las restricciones son expresiones lineales. Estos problemas pueden ser resueltos con el conocido método Simplex; sin embargo, muchos otros tipos de problemas de optimización son muy difíciles de resolver. De hecho, la mayor parte de los que podemos encontrar en la práctica entran dentro de esta categoría.

La idea intuitiva de problema "difícil de resolver" queda reflejada en el término científico *NP-hard* utilizado en el contexto de la complejidad algorítmica. En términos coloquiales podemos decir que un problema de optimización difícil es aquel para el que no podemos garantizar el encontrar la mejor solución posible en un tiempo razonable. La existencia de una gran cantidad y variedad de problemas difíciles, que aparecen en la práctica y que necesitan ser resueltos de forma eficiente, impulsó el desarrollo de procedimientos eficientes para encontrar buenas soluciones, aunque no fueran óptimas. Estos métodos, en los que la rapidez del proceso es tan importante como la calidad de la solución obtenida, se denominan heurísticos o aproximados. En [Díaz, 1996] y otros se recogen hasta ocho definiciones diferentes de algoritmo heurístico, entre las que se destaca la siguiente:

Un método heurístico es un procedimiento para resolver un problema de optimización bien definido mediante una aproximación intuitiva, en la que la estructura del problema se utiliza de forma inteligente para obtener una buena solución.

En contraposición a los *métodos exactos* que proporcionan una solución óptima del problema, como la programación dinámica y los algoritmos de Ramificación y Acotamiento (Branch and Bound) que para encontrar soluciones óptimas requieren tiempo superpolinomial a pesar de que estos algoritmos son más eficientes que la búsqueda exhaustiva pura, su tiempo de corrida es aún exponencial. Por esa razón es entonces natural buscar otra alternativa de solución, como son los algoritmos de aproximación que trabajan en tiempo polinomial. Los *métodos heurísticos* se limitan a proporcionar una buena solución no necesariamente óptima. [Adenso, 1996]

En la actualidad, la investigación se ha dirigido hacia el diseño de buenas heurísticas, es decir, algoritmos eficientes con respecto al tiempo de cómputo y al espacio de memoria, y con cierta verosimilitud de entregar una solución "buena" esto es, relativamente cercana a la óptima mediante el examen de solo un pequeño subconjunto de soluciones del número total.

Los métodos descritos en este capítulo reciben el nombre de algoritmos, metaheurísticos o sencillamente heurísticos. Este término deriva de la palabra griega *heuriskein* que significa encontrar o descubrir y se usa en el ámbito de la optimización para describir una clase de algoritmos de resolución de problemas.
Aunque en un primer momento no fueron bien vistos en los círculos académicos acusados de escaso rigor matemático ,su interés práctico como herramienta útil que da soluciones a problemas reales, les fue abriendo po co a poco las puertas, sobre todo a partir de la mitad de los años sesenta con la proliferación de resultados en el campo de la complejidad computacional. [Adenso, 1996]

3.2 Fundamentos de los algoritmos heurísticos

En este sentido consideraremos los llamados problemas de *Optimización Combinatoria*. Ya se veía en el capítulo 1, que en estos problemas el objetivo es encontrar el máximo (o el mínimo) de una determinada función sobre un conjunto finito de soluciones que denotaremos por *S*. No se exige ninguna condición o propiedad sobre la función objetivo o la definición del conjunto *S*. Es importante notar que dada la finitud de *S*, las variables han de ser discretas, restringiendo su dominio a una serie finita de valores. Habitualmente, el número de elementos de *S* es muy elevado, haciendo impracticable la evaluación de todas sus soluciones para determinar el óptimo.
En los últimos años ha habido un crecimiento espectacular en el desarrollo de procedimientos heurísticos para resolver problemas de optimización.
Este hecho queda claramente reflejado en el gran número de artículos publicados en revistas especializadas. En 1995 se edita el primer número de la revista *Journal of Heuristics* dedicada íntegramente a la difusión de los procedimientos heurísticos. [Martí, 2001]

Existen varias razones para utilizar métodos heurísticos, entre las que podemos destacar: [Adenso, 1996]

- El problema es de una naturaleza tal que no se conoce ningún método exacto para su resolución. Ofrecer entonces una solución que sólo sea aceptablemente buena resulta de interés frente a la alternativa de no tener ninguna solución en absoluto.
- Aunque existe un método exacto para resolver el problema, su uso es computacionalmente muy costoso.
- El método heurístico es más flexible que un método exacto, permitiendo, por ejemplo, la incorporación de condiciones de difícil modelización.
- El método heurístico se utiliza como parte de un procedimiento global que garantiza el óptimo de un problema.

Existen dos posibilidades:

- El método heurístico proporciona una buena solución inicial de partida.
- El método heurístico participa en un paso intermedio del procedimiento, como por ejemplo las reglas de selección de la variable a entrar en la base en el método Simplex.

Al abordar el estudio de los algoritmos heurísticos podemos comprobar que dependen en gran medida del problema concreto para el que se han diseñado. En otros métodos de resolución de propósito general, como pueden ser los algoritmos exactos de Ramificación y Acotación, existe un procedimiento conciso y preestablecido, independiente en gran medida

del problema abordado. En los métodos heurísticos esto no es así. Las técnicas e ideas aplicadas a la resolución de un problema son específicas de éste y aunque, en general, pueden ser trasladadas a otros problemas, han de particularizarse en cada caso.

En las siguientes secciones segunda y tercera se describen los métodos heurísticos que podríamos denominar "clásicos".

Existen muchos métodos heurísticos de naturaleza muy diferente, por lo que es complicado dar una clasificación completa. Además, muchos de ellos han sido diseñados para un problema específico sin posibilidad de generalización o aplicación a otros problemas similares.

3.2.1 Clasificación de Algoritmos Heurísticos

El siguiente esquema trata de dar unas categorías amplias, no excluyentes, en donde ubicar a los heurísticos más conocidos: [Adenso, 1996]

3.2.1.1 Métodos de descomposición: El problema original se descompone en subproblemas más sencillos de resolver, teniendo en cuenta, aunque sea de manera general, que ambos pertenecen al mismo problema. Algunos autores [Ball,Magazine, 1981] diferencian entre métodos de descomposición (los problemas se resuelven en cascada) y métodos de partición (cuando los subproblemas son independientes entre sí)

3.2.1.2 Métodos Inductivos: La idea de estos métodos es generalizar de versiones pequeñas o más sencillas al caso completo. Propiedades o técnicas identificadas en estos casos más fáciles de analizar pueden ser aplicadas al problema completo.

3.3.1.3 Métodos de Reducción: Consiste en identificar propiedades que se cumplen mayoritariamente por las buenas soluciones e introducirlas como restricciones del problema. El objeto es restringir el espacio de soluciones simplificando el problema. El riesgo obvio es dejar fuera las soluciones óptimas del problema original.

3.3.1.4 Métodos Constructivos: Consisten en construir literalmente paso a paso una solución del problema. Usualmente son métodos deterministas y suelen estar basados en la mejor elección en cada iteración. Estos métodos han sido los más utilizados en el PAV.

3.3.1.5 Métodos de búsqueda local: A diferencia de los métodos anteriores, los procedimientos de búsqueda o mejora local comienzan con una solución del problema y la mejoran progresivamente. El procedimiento realiza en cada paso un movimiento de una solución a otra con mejor valor. El método finaliza cuando, para una solución, no existe ninguna otra accesible que la mejore.

Si bien todos estos métodos han contribuido a ampliar nuestro conocimiento para la resolución de problemas reales, los **métodos constructivos** y los de **búsqueda local** constituyen la **base de los procedimientos metaheurísticos**.
Por ello, es que se estudian en éste capítulo los métodos constructivos y de búsqueda local de manera más detallada.
Se podrán encontrar alusiones a lo largo del texto a cualquiera de los métodos de descomposición, inductivos o de reducción, pero no se dedicará una sección específica a su estudio.

Alternativamente, se presentará especial atención a los métodos resultantes de combinar la construcción con la búsqueda local y sus diferentes variantes puesto que puede considerarse un punto de inicio en el desarrollo de método metaheurísticos.

En los últimos años han aparecido una serie de métodos bajo el nombre de **Metaheurísticos** con el propósito de obtener mejores resultados que los alcanzados por los heurísticos tradicionales. El termino metaheurístico fue introducido por Fred Glover en 1986. En este trabajo se utilizará la acepción de heurísticos para referirnos a los métodos clásicos en contraposición a la de metaheurísticos que se reservan para los más recientes y complejos.

En algunos textos podemos encontrar la expresión "heurísticos modernos" refiriéndose a los metaheurísticos [Reeves, 1995]. En [Osman y Kelly ,1996] se introduce la siguiente definición:

Los procedimientos Metaheurísticos son una clase de métodos aproximados que están diseñados para resolver problemas difíciles de optimización combinatoria, en los que los heurísticos clásicos no son efectivos. Los Metaheurísticos proporcionan un marco general para crear nuevos algoritmos híbridos, combinando diferentes conceptos derivados de la inteligencia artificial, la evolución biológica y los mecanismos estadísticos.

Los procedimientos Meta-Heurísticos se sitúan conceptualmente "por encima" de los heurísticos en el sentido que guían el diseño de éstos. Así, al enfrentarnos a un problema de optimización, podemos escoger cualquiera de estos métodos para diseñar un algoritmo específico que lo resuelva aproximadamente.

En estos momentos existe un gran desarrollo y crecimiento de estos métodos. Por lo que se puede convertir en una gran tarea el clasificar y agrupar las distintas técnicas que se han desarrollado, así, en este trabajo se limitará a aquellos procedimientos relativamente consolidados y que han probado su eficacia sobre una colección significativa de problemas. Específicamente se considerará la Búsqueda Tabú (Tabú Search TS), el Recocido Simulado (Templado Simulado) y los algunos Métodos Evolutivos, incluyendo los Algoritmos Genéticos (AG) y la Búsqueda Dispersa (BD).

También se incluyen algunos de los métodos GRASP (Greedy Randomized Adatative Search Procedure) junto con los Multi-Arranque que han sido incluidos en el apartado de Métodos Combinados que sirve de "puente" entre los métodos heurísticos y los metaheurísticos.

Al resolver un problema de forma heurística debemos de medir la calidad de los resultados puesto que, como ya hemos mencionado, la optimalidad no está garantizada. Por lo que en la siguiente sección se muestran los principales métodos para medir la calidad y eficiencia de un algoritmo y determinar su valía frente a otros.

3.2.2 Medidas de calidad de un algoritmo

Un buen algoritmo heurístico debe de tener las siguientes propiedades:

1. **Eficiente**: Un esfuerzo computacional realista para obtener la solución.
2. **Bueno**: La solución debe de estar, en promedio, cerca del óptimo.
3. **Robusto**: La probabilidad de obtener una mala solución (lejos del óptimo) debe ser baja.

Para medir la calidad de un heurístico existen diversos procedimientos, entre los que se encuentran los siguientes:

3.2.2.1 Comparación con la solución óptima

Aunque normalmente se recurre al algoritmo aproximado por no existir un método exacto para obtener el óptimo, o por ser éste computacionalmente muy costoso, en ocasiones puede que dispongamos de un procedimiento que proporcione el óptimo para un conjunto limitado de ejemplos (usualmente de tamaño reducido). Este conjunto de ejemplos puede servir para medir la calidad del método heurístico.

Normalmente se mide, para cada uno de los ejemplos, la desviación porcentual de la solución heurística frente a la óptima, calculando posteriormente el promedio de dichas desviaciones. Si llamamos c_h al costo de la solución del algoritmo heurístico y c_{opt} al costo de la solución óptima de un ejemplo dado, en un problema de minimización la desviación porcentual viene dada por la expresión: $\frac{c_h - c_{opt}}{c_{opt}} \cdot 100$.

3.2.2.2 Comparación con una cota

En ocasiones el óptimo del problema no está disponible ni siquiera para un conjunto limitado de ejemplos. Un método alternativo de evaluación consiste en comparar el valor de la solución que proporciona el heurístico con una cota del problema (inferior si es un problema de minimización y superior si es de maximización). Obviamente la bondad de esta medida dependerá de la bondad de la cota (cercanía de ésta al óptimo), por lo que, de alguna manera, tendremos que tener información de lo buena que es dicha cota. En caso contrario la comparación propuesta no tiene demasiado interés. [Osman y Kelly ,1996]

3.2.2.3 Comparación con un método exacto truncado

Un método enumerativo como el de Ramificación y Acotación explora una gran cantidad de soluciones, aunque sea únicamente una fracción del total, por lo que los problemas de grandes dimensiones pueden resultar computacionalmente inabordables con estos métodos. Sin embargo, podemos establecer un límite de iteraciones (o de tiempo) máximo de ejecución para el algoritmo exacto. También podemos saturar un nodo en un problema de maximización, cuando su cota inferior sea menor o igual que la cota superior global más un cierto valor *a* (análogamente para el caso de minimizar). De esta forma se garantiza que el valor de la mejor solución proporcionada por el procedimiento no dista más de *a* del valor óptimo del problema. En cualquier caso, la mejor solución encontrada con estos procedimientos truncados proporciona una cota con la que contrastar el heurístico. [Osman y Kelly ,1996]

3.2.2.4 Comparación con otros heurísticos

Este es uno de los métodos más empleados en problemas difíciles (NP-*hard*) sobre los que se ha trabajado durante tiempo y para los que se conocen algunos buenos heurísticos. Al igual que ocurre con la comparación con las cotas, la conclusión de dicha comparación está en función de la bondad del heurístico escogido.

3.2.2.5 Análisis del peor caso

Uno de los métodos que durante un tiempo tuvo bastante aceptación es analizar el comportamiento en el peor caso del algoritmo heurístico; esto es, considerar los ejemplos que sean más desfavorables para el algoritmo y acotar analíticamente la máxima desviación respecto del óptimo del problema.
Lo mejor de este método es que acota el resultado del algoritmo para cualquier ejemplo; sin embargo, por esto mismo, los resultados no suelen ser representativos del comportamiento medio del algoritmo. Además, el análisis puede ser muy complicado para los heurísticos más sofisticados.
Aquellos algoritmos que, para cualquier ejemplo, producen soluciones cuyo costo no se aleja de un porcentaje ε del costo de la solución óptima, se llaman *Algoritmos ε Aproximados.* Esto es; en un problema de minimización se tiene que cumplir para un $\varepsilon \succ 0$ que: $c_h \leq (1+\varepsilon)c_{opt}$

En este sentido se define el cociente de aproximación de un algoritmo *A*, denotado por C_A, para una instancia I de un problema de optimización $\prod(con|I|=n)$ como:

$$C_A = \min\left\{\frac{V(I,A(I))}{OPT(I)}, \frac{OPT(I)}{V(I,A(I)}\right\}$$

Nótese que C_A es siempre menor o igual a 1 y el factor de garantía r_A del algoritmo A para $\prod$ será:
A una solución que está dentro de un factor multiplicativo r_A del valor óptimo se le conoce como una r_A-aproximación, y decimos que:
Un problema NPO es aproximable dentro de un factor r_A si éste tiene un algoritmo de aproximación de tiempo polinomial con factor de garantía r_A .

Donde un problema NPO es una clase de problemas de optimización que se derivan de problemas de decisión en NP y son llamados NPO, como un acrónimo para designar que son problemas de optimización derivados de problemas NP.

Se dice que un algoritmo A es un algoritmo de aproximación-α para un problema de optimización $\prod$, con α una constante. Si A es un algoritmo de aproximación tal que para toda instancia I de $\prod$, produce una solución que está dentro de α-veces el OPT(I).[Castañeda, 2000]

También es usual considerar el término algoritmo de aproximación-α, para algoritmos aleatorios de tiempo polinomial que proporcionan soluciones cuyo valor esperado es al menos α-veces el óptimo. A α se le llama la constante de aproximación o bien la eficiencia garantizada que proporciona el algoritmo A.
Es por eso que los problemas se clasifican según su factor de aproximación en:

a) Problemas que no pueden aproximarse dentro de ningún factor constante α.
b) Problemas de optimización que se ha demostrado poseen un factor constante de aproximación; y que a pesar del trabajo realizado, no se han podido mejorar tales factores. [Castañeda,2000]

En este sentido, si tenemos que un problema $\prod$ esta en la clase *APX* (Problema de Aproximación X) si existe un algoritmo de tiempo polinomial para $\prod$ cuyo factor de aproximación o garantía está acotado por una constante.

Por ejemplo, dentro de los problemas NP-Completos se encuentra el PAV, que cuando el costo de las aristas satisface la desigualdad del triángulo (Δ PAV) existe un algoritmo simple que siempre produce una ruta de longitud cuando más dos veces la longitud de la ruta óptima. Esto tiene un tiempo de ejecución de $O(n^2)$. Un algoritmo mejorado con tiempo de corrida de $O(n^4)$ siempre encuentra una ruta con constante de aproximación de a lo más 3/2 de longitud de la ruta óptima. Y hasta la fecha no se conoce una mejor aproximación del algoritmo.

Sea f una función, $f-APX$ denota la clase de problemas en NPO que son aproximables dentro de un factor f, así, se obtiene una jerarquía de clases de complejidad.
Por ejemplo, $poly-APX$ y $\log-APX$ son las clases de problemas en NPO, los cuales tienen, respectivamente, algoritmos de aproximación con un factor de aproximación o garantía acotado polinomialmente y logarítmicamente, con respecto a la longitud de la entrada .

3.3 Métodos constructivos en el problema del viajante

Los métodos constructivos son procedimientos iterativos que, en cada paso, añaden un elemento hasta completar una solución. Usualmente son métodos deterministas y están basados en seleccionar, en cada iteración, el elemento con mejor evaluación. Estos métodos son muy dependientes del problema que resuelven. En esta sección se describen cuatro de los métodos más conocidos para el PAV, tal y como aparecen en la revisión realizada por: [Jünger, Reinelt , Rinaldi ,1995]

3.3.1 Heurísticos del vecino más próximo o más cercano

Uno de los heurísticos más sencillos para el PAV es el llamado "del vecino más cercano", que trata de construir un ciclo Hamiltoniano de bajo costo, basándose en el vértice cercano a uno dado. Este algoritmo es debido a Rosenkrantz, Stearns y Lewis que lo desarrollaron en el año de 1977; y su código, en una versión estándar, es el siguiente:

Algoritmo del *vecino más Próximo o más cercano*

Inicialización

Seleccionar un vértice j al azar.

Hacer $t = j$ y $W = V \setminus \{j\}$

Mientras $(W \neq 0)$

Tomar $j \in W / c_{tj} = \min\{c_{ti} / i \in W\}$

Conectar t a j

Hacer $W = W \setminus \{j\} t = j$

Figura 3.1 Algoritmo del vecino más Próximo o más cercano

Este procedimiento realiza un número de operaciones de orden $O(n^2)$. Si seguimos la evolución del algoritmo al construir la solución de un ejemplo dado, veremos que comienza muy bien, seleccionando aristas de bajo costo.
[Jünger, Reinelt , Rinaldi ,1995]

Sin embargo, al final del proceso probablemente quedarían vértices cuya conexión obligaría a introducir aristas de costo elevado. Esto es lo que se conoce como *miopía* del procedimiento, ya que, en una iteración escoge la mejor opción disponible sin"ver" que esto puede obligar a realizar malas elecciones en iteraciones posteriores. [Martí, 2001]

El algoritmo tal y como aparece puede ser programado en unas pocas líneas de código. Sin embargo, una implementación directa sería muy lenta al ejecutarse sobre ejemplos de gran tamaño (10000 vértices). Así pues, incluso para un heurístico tan sencillo como éste, es importante pensar en la eficiencia y velocidad de su código.

Para reducir la miopía del algoritmo y aumentar su velocidad se introduce el concepto de subgrafo candidato, junto con algunas modificaciones en la exploración. Un subgrafo candidato es un subgrafo del grafo completo con los *n* vértices y únicamente las aristas consideradas "atractiva" para aparecer en un ciclo Hamiltoniano de bajo costo. Una posibilidad es tomar, por ejemplo, el subgrafo de los *k* vecinos más cercanos; esto es, el subgrafo con los *n* vértices y para cada uno de ellos las aristas que lo unen con los *k* vértices más cercanos. Este subgrafo también será usado en otros procedimientos.

El algoritmo puede "mejorarse" en los siguientes aspectos:

- Para seleccionar el vértice j que se va a unir a t (y por lo tanto al tour parcial en construcción), en lugar de examinar todos los vértices, se examinan únicamente los adyacentes a t en el subgrafo candidato. Si todos ellos están ya en el tour parcial, entonces sí que se examinan todos los posibles.
- Cuando un vértice queda conectado (con grado 2) al tour en construcción, se eliminan del subgrafo candidato las aristas incidentes con él.
- Se especifica un número $s < k$ de modo que cuando un vértice que no está en el tour está conectado únicamente a *s* o menos aristas del subgrafo candidato, se considera que se está quedando aislado. Por ello se inserta inmediatamente en el tour. Como punto de inserción se toma el mejor de entre los k vértices más cercanos presentes en el tour.

3.3.2 Heurísticos de Inserción

Otra aproximación intuitiva a la resolución del PAV consiste en comenzar construyendo ciclos que visiten únicamente unos cuantos vértices, para posteriormente extenderlos insertando los vértices restantes. En cada paso se inserta un nuevo vértice en el ciclo hasta obtener un ciclo Hamiltoniano.

Este procedimiento se debe a los mismos autores que el anterior y su esquema es el siguiente:

Algoritmo *de Inserción*

Inicialización

Seleccionar un ciclo inicial (subtour) con k vértices.

Hacer $W \equiv V \setminus$ {vértices seleccionados}

Mientras $(W \neq 0)$

Tomar $j \in W$ de acuerdo con algún criterio preestablecido

Insertar j donde menos incremente la longitud del ciclo

Hacer $W = W \setminus \{j\}$.

Figura 3.2 Algoritmo de inserción

Existen varias posibilidades para implementar el esquema anterior de acuerdo con el criterio de selección del vértice *j* de *W* a insertar en el ciclo. Se define la distancia de un vértice *v* al ciclo como el mínimo de las distancias de *v* a todos los vértices del ciclo:

$$d_{\min}(v) = \min\{c_{iv} / i \in V \setminus W\}$$

Los criterios más utilizados son:

Inserción más cercana: Seleccionar el vértice *j* más cercano al ciclo.

$$d_{\min}(j) = \min\{d_{\min}(v) / v \in W\}$$

Inserción más lejana: Seleccionar el vértice *j* más lejano al ciclo.

$$d_{\min}(j) = \max\{d_{\min}(v) / v \in W\}$$

Inserción más barata: Seleccionar el vértice *j* que será insertado con el menor incremento del costo.

Inserción aleatoria: Seleccionar el vértice *j* al azar.

La figura 3.3 muestra la diferencia entre estos criterios en un caso dado. El ciclo actual está formado por 4 vértices y hay que determinar el próximo a insertar. La inserción más cercana escogería a el vértice *i*, la más lejana el *s* y la más barata el *k*.

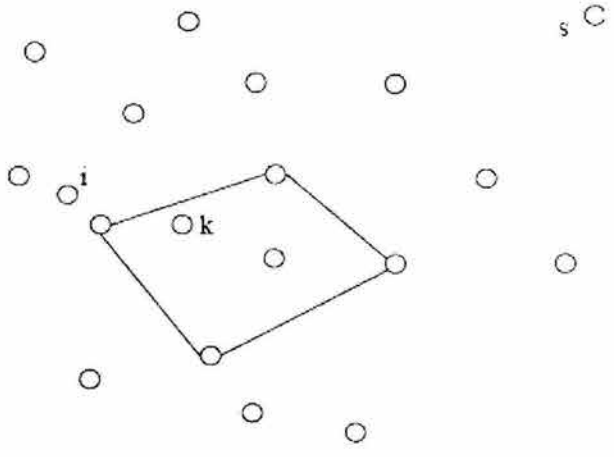


Figura 3.3 Selección del vértice a insertar

Todos los métodos presentan un tiempo de ejecución de $O(n^2)$ excepto la inserción más cercana que es de $O(n^2 \log n)$. [Jünger, Reinelt , Rinaldi ,1995]

3.3.3 Heurísticos basados en árboles generadores o árboles de expansión

Los heurísticos considerados anteriormente construyen un ciclo Hamiltoniano basándose únicamente en los costos de las aristas. Los heurísticos de este apartado se basan en el árbol generador de costo mínimo, lo que aporta una información adicional sobre la estructura del grafo. Para entender en mayor medida este apartado es necesario recordar los siguientes conceptos ya vistos en el capítulo 1.

- Un grafo es conexo, si todo par de vértices está unido por un camino.
- Un árbol es un grafo conexo que no contiene ciclos. El numero de aristas de un árbol es igual al numero de vértices menos uno.
- Un árbol generador de un grafo $G = (V, A, C)$ es un árbol sobre todos los vértices y tiene, por tanto, $|V| - 1$ aristas de G.
- Un árbol generador de mínimo peso (o de costo mínimo) es aquel que de entre todos los árboles generadores de un grafo dado, presenta la menor suma de los costos de sus aristas.
- Un acoplamiento de un grafo $G = (V, A, C)$ es un subconjunto M del conjunto A de aristas cumpliendo que cada vértice del grafo es, a lo sumo, incidente con una arista de M.
- Un acoplamiento sobre un grafo $G = (V, A, C)$ es perfecto, si es de cardinalidad máxima e igual a $\lceil |V|/2 \rceil$.

Las figuras siguientes ilustran los conceptos vistos sobre un grafo completo de 8 vértices. En la figura 3.4 tenemos un árbol generador. Notesé que contiene a todos los vértices y no hay ningún ciclo. La figura 3.5 muestra un acoplamiento perfecto en el que podemos ver cómo cada vértice es incidente con una, y sólo una, de las aristas. Al ser un grafo de 8 vértices el número máximo de aristas en un acoplamiento es de 4, por lo que el de la figura es perfecto.

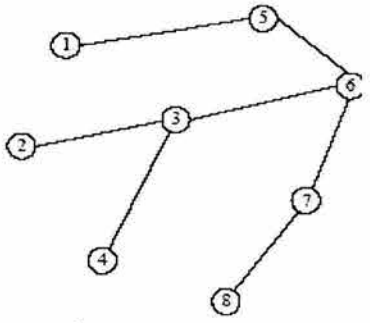


Figura 3.4 Árbol generador

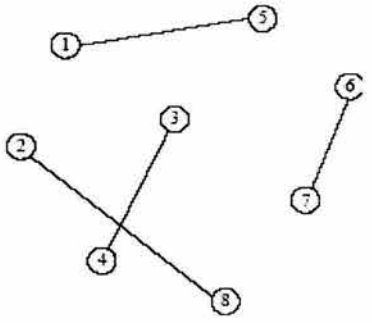


Figura 3.5 Acoplamiento Perfecto

El algoritmo debido a Prim realizado en 1957, obtiene un árbol de expansión mínimo de un grafo G completo. El algoritmo comienza por definir el conjunto T de aristas del árbol (inicialmente vacío) y, el conjunto U de vértices del árbol (inicialmente formado por uno elegido al azar). En cada paso se calcula la arista de menor costo que une U con $V \setminus U$

añadiéndola a T y pasando su vértice adyacente de $V \setminus U$ a U. El procedimiento finaliza cuando U es igual a U ; en cuyo caso el conjunto T proporciona la solución.

Dado un ciclo $v_{i0}, v_{i1}, ..., v_{ik}$ que pasa por todos los vértices de G (no necesariamente simple), el siguiente procedimiento obtiene un ciclo Hamiltoniano comenzando en v_{i0} y terminando en $(v_{i0} = v_{ik})$. En el caso de grafos con costos cumpliendo la desigualdad triangular (como es el caso de los grafos con distancias euclídeas), este procedimiento obtiene un ciclo de longitud menor o igual que la del ciclo de partida.

Algoritmo *de Obtención deTour*

Inicialización
Seleccionar un ciclo inicial (subtour) con k vértices.
Hacer $T = \{v_{i0}\}$, $v = v_{i0}$ y $s = 1$
Mientras $(|T| \prec |V|)$
Si v_{is} no en T hacer $T = T \cup \{v_{is}\}$, conectar v a v_{is} y hacer $v = v_{is}$
Hacer $s = s + 1$
Conectar v a v_{i0} y formar el ciclo Hamiltoniano.

Figura 3.6 Algoritmo de Obtención de Tour

A partir de los elementos descritos se puede diseñar un algoritmo para obtener un ciclo Hamiltoniano. Basta con construir un árbol de expansión mínimo (figura 3.7), considerar el ciclo en el que todas las aristas del árbol son recorridas dos veces, cada vez en un sentido (figura 3.8), y aplicar el *algoritmo de obtención de un tour* a dicho ciclo (figura 3.9). El ejemplo de las figuras mencionadas ilustra dicho procedimiento sobre un grafo completo con 10 vértices.

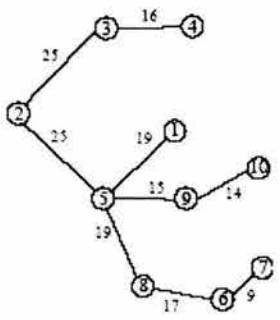


Figura 3.7. Árbol generador

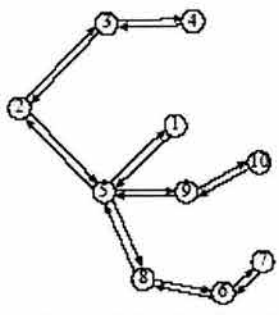
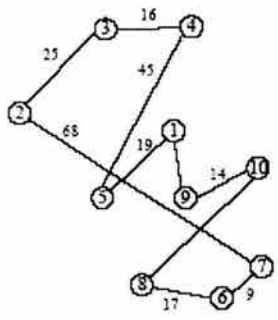


Figura 3.8. Duplicación



Figuras 3.9.

La Figura 3.7 muestra un árbol de expansión mínimo o árbol generador de mínimo peso igual a 156. En la Figura 3.8 se han duplicado las aristas y se señala mediante flechas la dirección del ciclo resultante. Su costo obviamente será de 156 *X* 2 = 312. Aplicando el procedimiento de obtención de tour al ciclo de la figura 3.8, se obtiene el ciclo Hamiltoniano de la figura 3. 9 con un costo de 258.

El proceso de duplicación de las aristas (recorrerlas todas en ambos sentidos) para obtener un tour aumenta en gran medida el costo de la solución. Podemos ver que es posible obtener un ciclo que pase por todos los vértices de G a partir de un árbol generador, sin necesidad de duplicar todas las aristas. De hecho, basta con añadir aristas al árbol de modo que todos los vértices tengan grado par.

3.3.3.1 Algoritmo de Christofides

El siguiente procedimiento, debido a Christofides, quien lo desarrolla en 1976, calcula un acoplamiento perfecto de mínimo peso sobre los vértices de grado impar del árbol de expansión.

Añadiendo al árbol las aristas del acoplamiento se obtiene un grafo con todos los vértices pares, y por lo tanto, un ciclo del grafo original, a partir del cual ya se vio cómo obtener un tour.

Algoritmo *de Christofides*

1. Calcular un árbol de expansión mínimo.
2. Obtener el conjunto de vértices de grado impar en el árbol.
3. Obtener un acoplamiento perfecto de mínimo peso sobre dichos vértices.
4. Añadir las aristas del acomplamiento al árbol.
5. Aplicar el procedimiento de obtención de Tour.

Figura 3.10 Algoritmo de Christofides

El cálculo del acoplamiento perfecto de costo mínimo sobre un grafo de k vértices se realiza en un tiempo $O(k^3)$ con el algoritmo que se encuentra en [Edmonds, 1965]. Dado que un árbol generador de mínimo peso tiene como máximo n-1 hojas (vértices de grado 1 en el árbol), el procedimiento de Christofides tendrá un tiempo de orden $O(n^3)$.

Propiedad: El algoritmo de Christofides sobre ejemplos cuya matriz de distancias cumple la desigualdad triangular produce una solución cuyo valor es cuando más, 1.5 veces el valor óptimo:

$$c_h \leq \frac{3}{2} c_{OPT}$$

Es decir, es un algoritmo 1/2 - aproximado sobre esta clase de ejemplos.

Prueba: Sea $c(AGMP)$ el costo del árbol generador de mínimo peso y $c(A)$ el costo del acoplamiento perfecto calculado en el algoritmo de Christofides. Al añadir las aristas del acoplamiento al árbol se obtiene un ciclo (con posibles repeticiones) cuyo costo es la suma de ambos costos. Dado que la matriz de distancias cumple la desigualdad triangular, al aplicar el algoritmo de obtención de tour el costo puede reducirse eventualmente. Por ello el costo de la solución obtenida, cH , cumple:

$$c_H \leq c(AGMP) + c(A)$$

Un árbol de expansión mínimo, por construcción, tiene un costo menor que cualquier ciclo Hamiltoniano y, por lo tanto, que el ciclo Hamiltoniano de costo mínimo (tour óptimo). Para probarlo basta con considerar un ciclo Hamiltoniano y quitarle una arista, con lo que se obtiene un árbol generador. El árbol de expansión mínimo tiene, obviamente, un costo menor que dicho árbol generador y, por lo tanto, que el ciclo Hamiltoniano. Luego:

$$c(AGMP) \leq c_{OPT}$$

El acoplamiento del paso 3 del algoritmo de Christofides tiene un costo menor o igual que la mitad de la longitud de un tour óptimo en un grafo con costos, cumpliendo la desigualdad triangular. Para probarlo consideremos un tour óptimo y llamemos S al conjunto de vértices de grado impar en el árbol. El algoritmo calcula un acoplamiento perfecto de costo mínimo sobre los vértices de S. La Figura 3.11 muestra un ciclo Hamiltoniano óptimo y los vértices de S en oscuro. Además aparecen dos acoplamientos perfectos sobre S, A_1 (trazo continuo) y A_2 (trazo discontinuo), en donde cada vértice está acoplado al más próximo en el tour óptimo.

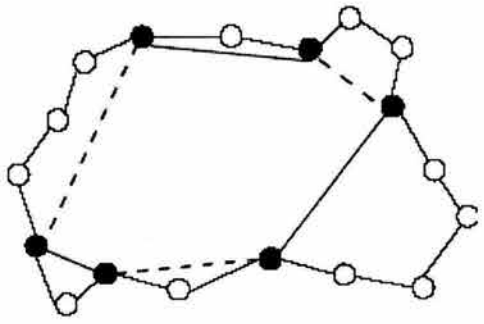


Figura 3.11 Dos acoplamientos sobre S

Como se cumple la desigualdad triangular, se tiene que $c(A_1)+c(A_2) \leq c_{OPT}$. Es evidente que por ser el de costo mínimo $c(A) \leq c(A_1)$ y $c(A) \leq c(A_2)$ de donde: $2c(A) \leq c_{OPT}$

Las figuras siguientes ilustran el método de Christofides sobre el mismo ejemplo de las figuras 3.7, 3.8 y 3.9. En la figura 3.12 aparecen oscurecidos los vértices de grado impar en el árbol de expansión, y en trazo discontinuo las aristas del acoplamiento perfecto de costo mínimo sobre tales vértices. Al añadirlas se obtiene un tour de costo 199. La figura 3.13 muestra el ciclo Hamiltoniano que se obtiene al aplicarle el procedimiento de obtención del ciclo al tour de la figura anterior. La solución tiene un costo de 203, mientras que la obtenida con el procedimiento de duplicar aristas (Figura 3.9) tenía un costo de 258.

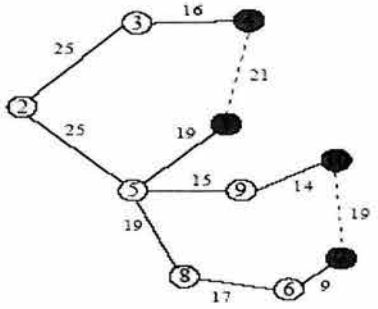


Figura 3.12. Acoplamiento Perfecto

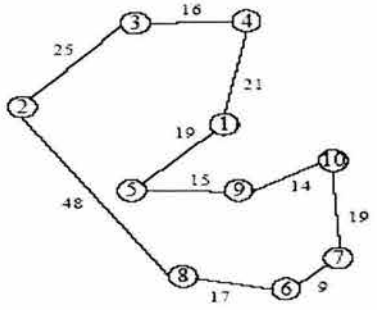


Figura 3.13 Ciclo Hamiltoniano

3.3.4 Heurísticos Basados en Ahorros

Los métodos de esta sección son debidos a Clarke y Wright desarrollados en 1964 y fueron propuestos inicialmente para problemas de rutas de vehículos. Veamos una adaptación de estos procedimientos al problema del agente viajero. El algoritmo siguiente se basa en combinar sucesivamente subtours hasta obtener un ciclo Hamiltoniano. Los subtours considerados tienen un vértice común llamado *base*. El procedimiento de unión de subtours se basa en eliminar las aristas que conectan dos vértices de diferentes subtours con el vértice base, uniendo posteriormente los vértices entre sí. Llamamos *ahorro* a la diferencia del costo entre las aristas eliminadas y la añadida.

Algoritmo *de Ahorros*

Inicialización
Tomar un vértice $z \in V$ como base
Establecer los $n-1$ subtours $\left[(z,v),(v,z)\right], \forall v \in V \setminus \{z\}$
Mientras (queden dos o más subtours)
Para cada par de subtours calcular el ahorro de unirlos al eliminar en cada una de las aristas que lo une con z y conectar los dos vértices asociados. Unir los dos subtours que produzcan un ahorro mayor.

Figura 3.14 Algoritmo de Ahorros

Las figuras 3.15 y 3.16 ilustran una iteración del procedimiento. Podemos ver cómo se combinan dos subtuors eliminando las aristas de los vértices i y j al vértice base z, e insertando la arista (i, j)

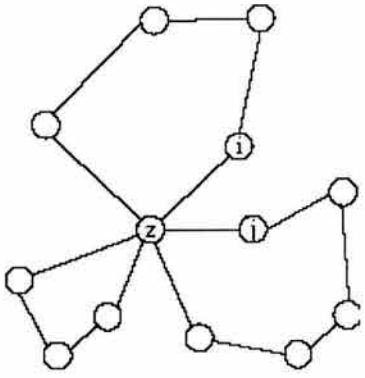


Figura 3.15 Subtours iniciales

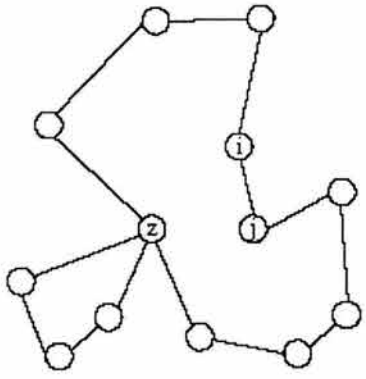


Figura 3.16 Subtours finales

En la implementación del algoritmo se tiene que mantener una lista con las combinaciones posibles. El punto clave de la implementación es la actualización de esta lista. Sin embargo, al unir dos subtours únicamente se ven afectados aquellos en los que su "mejor conexión" pertenece a alguno de los dos subtours recién unidos. Luego basta con actualizar éstos en cada iteración sin necesidad de actualizarlos todos cada vez que se realiza una unión.

Al igual que en otros heurísticos, podemos utilizar el subgrafo candidato (en el que están todos los vértices y sólo las aristas consideradas "atractivas") para acelerar los cálculos. Así, al actualizar la lista de las mejores conexiones únicamente se consideran aristas del subgrafo candidato. El método presenta un tiempo de ejecución de $O(n^3)$.

3.4 Búsqueda Local en el Problema del Agente Viajero

En este apartado se estudian diversos algoritmos basados en la búsqueda local. Al igual que ocurría con los métodos descritos en la sección anterior, estos algoritmos son muy dependientes del problema que resuelven. Específicamente, se considerarán tres de los métodos más utilizados, tal y como aparecen descritos en [Jünger, Reinelt y Rinaldi ,1995]. Se comenzará por definir y explicar algunos de los conceptos genéricos de estos métodos.
Los procedimientos de búsqueda local, también llamados de mejora, se basan en explorar el entorno o vecindad de una solución. Utilizan una operación básica llamada movimiento que, aplicada sobre los diferentes elementos de una solución, proporciona las soluciones de su entorno. Formalmente:
Definición: Sea X el conjunto de soluciones del problema combinatorio.
Cada solución x tiene un conjunto de soluciones asociadas $N(x) \subseteq X$ que denominaremos entorno de x.
Definición: Dada una solución x, cada solución de su entorno, $x' \in N(x)$, puede obtenerse directamente a partir de x mediante una operación llamada movimiento.

Un procedimiento de búsqueda local parte de una solución inicial x_0, calcula su entorno $N(x_0)$ y escoge una nueva solución x_1 en él. Dicho de otro modo, realiza el movimiento m_1 que aplicado a x_0 da como resultado x_1. Este proceso se aplica reiteradamente, describiendo una trayectoria en el espacio de soluciones.

Un procedimiento de búsqueda local queda determinado al especificar un entorno y el criterio de selección de una solución dentro del entorno. La definición de entorno/movimiento, depende en gran medida de la estructura del problema a resolver, así como de la función objetivo. También se pueden definir diferentes criterios para seleccionar una nueva solución del entorno. Uno de los criterios más simples consiste en tomar la solución con mejor evaluación de la función objetivo, siempre que la nueva solución sea mejor que la actual. Este criterio, conocido como *greedy*, permite ir mejorando la solución actual mientras se pueda. El algoritmo se detiene cuando la solución no puede ser mejorada. A la solución encontrada se le denomina *óptimo local* respecto al entorno definido.
El óptimo local alcanzado no puede mejorarse mediante el movimiento definido. Sin embargo, el método empleado no permite garantizar, de ningún modo, que sea el óptimo global del problema. Más aún, dada la "miopía" de la búsqueda local, es de esperar que en problemas de cierta dificultad, en general no lo sea.
La figura 3.17 muestra el espacio de soluciones de un problema de maximización de dos dimensiones donde la altura del grafico mide el valor de la función objetivo. Se considera un procedimiento de búsqueda local *greedy* iniciado a partir de una solución x_0 con valor 5 y que realiza 8 movimientos de mejora hasta alcanzar la solución x_8 con valor 13. La figura muestra cómo x_8 es un óptimo local y cualquier movimiento que se le aplique proporcionaré una solución con peor valor. Podemos ver cómo el óptimo global del problema, con un valor de 15, no puede ser alcanzado desde x_8, a menos que permitamos realizar movimientos que empeoren el valor de las soluciones y sepamos dirigir correctamente la búsqueda.

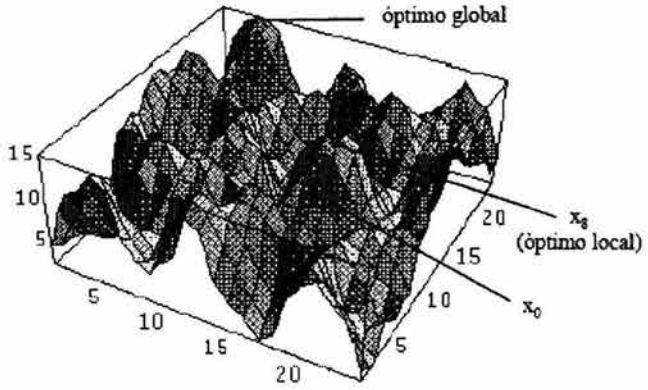


Figura 3.17 Óptimo local y global

Esta limitación de la estrategia *greedy* es el punto de partida de los procedimientos meta-heurísticos basados en búsqueda local: evitar el quedar atrapados en un óptimo local lejano del global. Para lo cual, como hemos visto, se hace preciso el utilizar movimientos que empeoren la función objetivo. Sin embargo esto plantea dos problemas. El primero es que al permitir movimientos de mejora y de no mejora, el procedimiento se puede ciclar, revisitando soluciones ya vistas, por lo que habría que introducir un mecanismo que lo impida. El segundo es que hay que establecer un criterio de parada ya que un procedimiento de dichas características podría iterar indefinidamente. Los procedimientos meta-heurísticos incorporan mecanismos sofisticados para solucionar eficientemente ambas cuestiones así como para tratar, en la medida de lo posible, de dirigir la búsqueda de forma inteligente.
Pese a la miopía de los métodos de búsqueda local simples, suelen ser muy rápidos y proporcionan soluciones que, en promedio, están relativamente cerca del óptimo global del problema. Además, dichos métodos suelen ser el punto de partida en el diseño de algoritmos meta-heurísticos más complejos.

En este apartado vamos a estudiar algunos métodos heurísticos de búsqueda local para el problema del viajante.

3.4.1 Procedimientos de 2- intercambio

Este procedimiento se basa en la siguiente observación para grafos con distancias euclídeas (o en general con costos cumpliendo la desigualdad triangular). Si un ciclo Hamiltoniano se cruza a sí mismo, puede ser fácilmente acortado, basta con eliminar las dos aristas que se cruzan y reconectar los dos caminos resultantes mediante aristas que no se corten. El ciclo final es más corto que el inicial.

Un *movimiento 2-opt* consiste en eliminar dos aristas y reconectar los dos caminos resultantes de una manera diferente para obtener un nuevo ciclo.
Las figuras 3.18 y 3.19 ilustran este movimiento en el que las aristas (i,j) y (l,k) son reemplazadas por (l,j) y (i,k). Notar que sólo hay una manera de reconectar los dos caminos formando un solo tour.

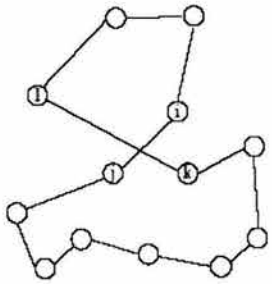


Figura 3.18 Solución original

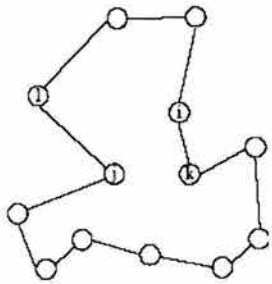


Figura 3.19 Solución mejorada

El siguiente código recoge el algoritmo heurístico de mejora 2-óptimo. Consiste en examinar todos los vértices, realizando, en cada paso, el mejor movimiento 2-opt asociado a cada vértice.

Algoritmo *2-óptimo*

Inicialización

Considerar un ciclo Hamiltoniano inicial

Para cada vértice i definir un conjunto de vértices $N(i)$

$move = 1$

Etiquetar todos los vértices como no explorados

Mientras (queden vértices por explorar)

Seleccionar un vértice i no explorado.

Examinar todos los movimientos 3-opt que eliminen 3 aristas teniendo cada una, al menos un vértice $N(i)$.

Si alguno de los movimientos examinados reduce la longitud del ciclo, realizar el mejor de todos y hacer $move = 1$. En otro caso etiquetar i como explorado.

Figura 3.20 Algoritmo 2-óptimo

La variable $move$ vale 0 si no se ha realizado ningún movimiento al examinar todos los vértices, y 1 en otro caso. El algoritmo finaliza cuando $move = 0$, con lo que queda garantizado que no existe ningún movimiento 2-opt que pueda mejorar la solución.

El orden en el que el algoritmo examina los nodos influye de manera notable en su funcionamiento. En una implementación sencilla podemos considerar el orden natural $1, 2, ..., n$. Sin embargo, es fácil comprobar que cuando se realiza un movimiento hay muchas posibilidades de encontrar movimientos de mejora asociados a los vértices que han intervenido en el movimiento recién realizado. Por ello, una implementación más eficiente consiste en considerar una lista de vértices candidatos a examinar. El orden inicial es el de los vértices en el ciclo comenzando por uno arbitrario y en cada iteración se examina el primero de la lista. Cada vez que se examina un vértice i, éste se coloca al final de la lista y, los vértices involucrados en el movimiento (vértices j, k y l de la figura 3.19) se insertan en primer lugar.

Dado que el proceso de examinar todos los movimientos 2-opt asociados a cada vértice es muy costoso computacionalmente, se pueden introducir las siguientes mejoras para acelerar el algoritmo:

- Exigir que al menos una de las dos aristas añadidas en cada movimiento, para formar la nueva solución, pertenezca al subgrafo candidato.

- Observando el funcionamiento del algoritmo se puede ver que en las primeras iteraciones la función objetivo decrece substancialmente, mientras que en las últimas apenas se modifica. De hecho la última únicamente verifica que es un óptimo local al no realizar ningún movimiento. Por ello, si interrumpimos el algoritmo antes de su finalización, ahorraremos bastante tiempo y no perderemos mucha calidad.

Es evidente que ambas mejoras reducen el tiempo de computación a expensas de perder la garantía de que la solución final es un óptimo local. Así pues, dependiendo del tamaño del ejemplo a resolver, así como de lo crítico que sea el tiempo de ejecución, se deben implementar o no.
El comprobar si existe, o no, un movimiento 2-opt de mejora utiliza un tiempo de orden $O(n^2)$, ya que se tienen que examinar todos los pares de aristas en el ciclo. Podemos encontrar clases de problemas para los que el tiempo de ejecución del algoritmo no está acotado polinómicamente.

3.4.2 Procedimientos de *k* – intercambio

Para introducir mayor flexibilidad al modificar un ciclo Hamiltoniano, podemos considerar el dividirlo en k partes, en lugar de dos, y combinar los caminos resultantes de la mejor manera posible. Llamamos movimiento *k-opt* a tal modificación. Es evidente que al aumentar *k* aumentará el tamaño del entorno y el número de posibilidades a examinar en el movimiento, tanto por las posibles combinaciones para eliminar las aristas del ciclo, como por la reconstrucción posterior. El número de combinaciones para eliminar k aristas en un ciclo viene dado por el número $\left(\frac{n}{k}\right)$.

Examinar todos los movimientos *k-opt* de una solución lleva un tiempo del orden de $o(n^k)$ por lo que, para valores altos de *k*, sólo es aplicable a ejemplos de tamaño pequeño.

En este apartado vamos a estudiar el caso de *k* = 3 y además impondremos ciertas restricciones para reducir el entorno y poder realizar los cálculos en un tiempo razonable.

En un movimiento 3-opt, una vez eliminadas las tres aristas hay ocho maneras de conectar los tres caminos resultantes para formar un ciclo. Las figuras siguientes ilustran algunos de los ocho casos. La figura 3.21 muestra el ciclo inicial en el que se encuentran las aristas $(a,b),(c,d)$ y (e,f) en las que se dividirá. La Figura 3.22 utiliza la propia arista (e,f) para reconstruir el ciclo, por lo que, este caso equivale a realizar un movimiento 2-opt sobre las aristas (a,b) y (c,d).

Análogamente podemos considerar los otros dos casos en los que se mantiene una de las tres aristas en el ciclo y se realiza un movimiento 2-opt sobre las restantes. Las Figuras 3.23 y 3.24 muestran un movimiento 3-opt "puro" en el que desaparecen del ciclo las tres aristas seleccionadas.

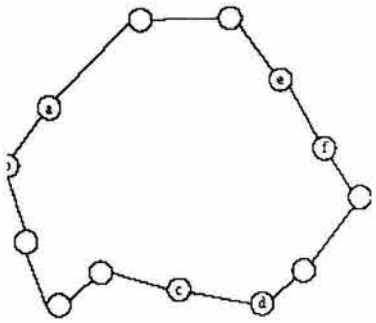


Figura 3.21 Ciclo inicial

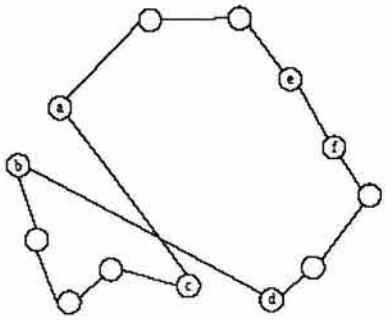


Figura 3.22 Movimiento 2-opt

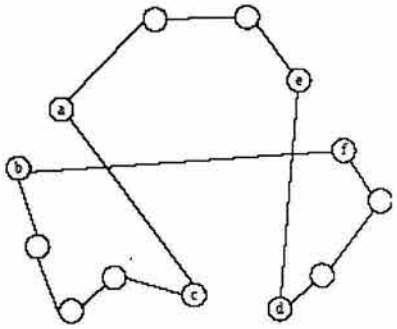


Figura 3.23 Movimiento 3-opt

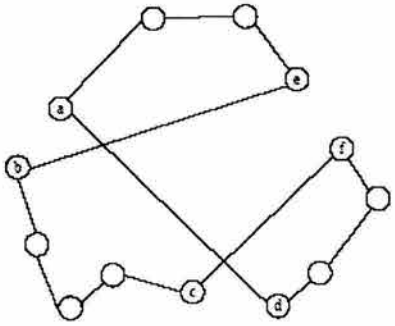


Figura 3.24 Movimiento 3-opt

A diferencia de los movimientos 2-opt, el reconstruir el ciclo una vez eliminadas las tres aristas es muy costoso. Notar que la dirección en el ciclo puede cambiar en todos los caminos menos en el más largo por lo que hay que realizar varias actualizaciones. Además, el mero hecho de examinar todas las posibilidades representa un esfuerzo computacional enorme. Por ello, se consideran únicamente algunos de los movimientos 3-opt.

En concreto se define para cada vértice *i* un conjunto de vértices $N(i)$ de modo que al examinar los movimientos 3-opt asociados a una arista (i, j), únicamente se consideran aquellos en los que las otras dos aristas tengan al menos uno de los vértices en $N(i)$. Una posibilidad para definir $N(i)$ consiste en considerar los vértices adyacentes a *i* en el subgrafo candidato.

Algoritmo 3-*óptimo restringido*

Inicialización
Considerar un ciclo Hamiltoniano inicial
Para cada vértice *i* definir un conjunto de vértices $N(i)$
move = 1
Etiquetar todos los vértices como no explorados

Mientras (queden vértices por explorar)
Seleccionar un vértice *i* no explorado.
Examinar todos los movimientos 3-opt que eliminen 3 aristas teniendo cada una, al menos un vértice $N(i)$.
Si alguno de los movimientos examinados reduce la longitud del ciclo, realizar el mejor de todos y hacer *move* = 1. En otro caso etiquetar *i* como explorado.

Figura 3.25 Algoritmo 3-óptimo restringido

3.4.3 Algoritmo de Lin y Kernighan

Como vimos en la introducción a los métodos de mejora (figura 3.17), el problema de los algoritmos de búsqueda local es que suelen quedarse atrapados en un óptimo local. Vimos que para alcanzar una solución mejor a partir de un óptimo local habría que comenzar por realizar movimientos que empeoren el valor de la solución, lo que conduciría a un esquema de búsqueda mucho más complejo, al utilizar el algoritmo tanto movimientos de mejora como de no mejora.

El algoritmo de Lin y Kernighan parte de este hecho y propone un movimiento compuesto, en donde cada una de las partes consta de un movimiento que no mejora necesariamente pero el movimiento compuesto si es de mejora. De esta forma es como si se realizaran varios movimientos simples consecutivos en donde algunos empeoran y otros mejoran el valor de la solución, pero no se pierde el control sobre el proceso de búsqueda ya que el movimiento completo sí que mejora. Además, combina diferentes movimientos simples, lo cual es una estrategia que ha producido muy buenos resultados en los algoritmos de búsqueda local. En concreto la estrategia denominada "cadenas de eyección" se basa en encadenar movimientos y ha dado muy buenos resultados en el contexto de la Búsqueda Tabú.

Se pueden considerar muchas variantes para este algoritmo. En este apartado se considerará una versión sencilla basada en realizar dos movimientos 2-opt seguidos de un movimiento de inserción. Se Ilustrará el procedimiento mediante el ejemplo desarrollado en las figuras siguientes sobre un grafo de 12 vértices. Consideramos el ciclo Hamiltoniano inicial dado por el orden natural de los vértices y lo representamos tal y como aparece en la Figura 3.26

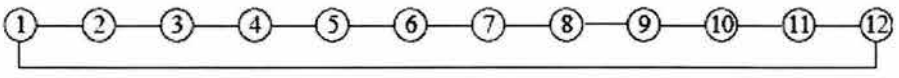


Figura 3.26

Paso 1: Realiza un movimiento 2-opt reemplazando las aristas *(12,1)* y *(5,6)* por *(12,5)* y *(1,6)*. El resultado se muestra en la figura 3.27

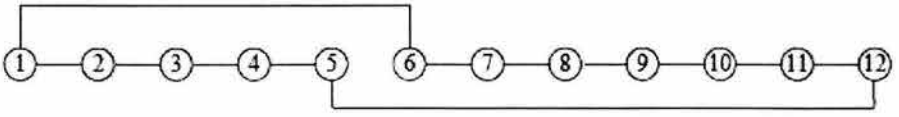


Figura 3.27

Paso 2: Realiza un movimiento 2-opt reemplazando las aristas *(6,1)* y *(3,4)* por *(6,3)* y *(1,4)*. Ver Ver figura 3.28

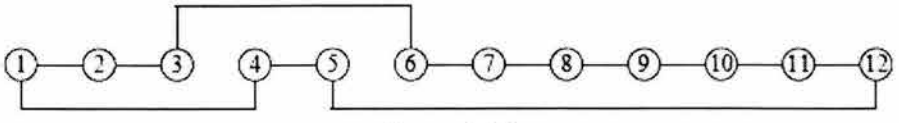


Figura 3.28

Paso 3: Realiza un movimiento de inserción, insertando el vértice 9 entre el 1 y el 4 (Figura 3.29).

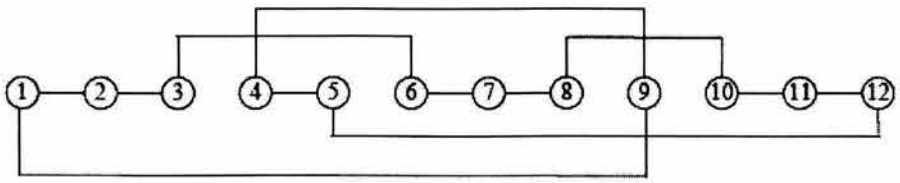


Figura 3.29

El algoritmo funciona de igual modo que el 2-óptimo o el 3-óptimo: parte de un ciclo Hamiltoniano inicial y realiza movimientos de mejora hasta alcanzar un óptimo local.

Algoritmo *de Lin y Kernighan*

Inicialización
  Considerar un ciclo Hamiltoniano inicial
  $move = 1$
Mientras ($move = 1$)
  $move = 0$
  Etiquetar todos los vértices como no explorados
  Mientras (queden vértices por explorar)
    Seleccionar un vértice i no explorado.
    Examinar todos los movimientos (2-opt, 2-opt, inserción) que incluyan la arista de i a su sucesor en el ciclo.
  Si alguno de los movimientos examinados reduce la longitud del ciclo, realizar el mejor de todos y hacer $move = 1$. En otro caso etiquetar i como explorado.

Figura 3.30 Algoritmo de Lin y Kernighan

Dado el gran número de combinaciones posibles para escoger los movimientos, es evidente que una implementación eficiente del algoritmo tendría que restringir el conjunto de movimientos a examinar en cada paso. De entre las numerosas variantes estudiadas para reducir los tiempos de computación del algoritmo y aumentar la eficiencia del proceso, destacamos las dos siguientes:

- Utilizar el subgrafo candidato en el que únicamente figuran las aristas relativas a los 6 vecinos más cercanos para cada vértice. Se admiten movimientos compuestos de hasta 15 movimientos simples todos del tipo 2-opt o inserción. Para el primer movimiento simple únicamente se examinan 3 candidatos.

- El subgrafo candidato está formado por las aristas relativas a los 8 vecinos más cercanos. Se admiten hasta 15 movimientos simples del tipo 2-opt o inserción por cada movimiento completo. En los 3 primeros movimientos simples únicamente se examinan 2 aristas.

3.5 Métodos combinados

En los apartados se han visto los métodos constructivos que obtienen una solución del problema y los métodos de mejora que, a partir de una solución inicial, tratan de obtener nuevas soluciones con mejor valor. Es evidente que ambos métodos pueden combinarse, tomando los segundos como solución inicial la obtenida con los primeros. En este apartado se estudiarán algunas variantes y mejoras sobre tal esquema.

Como se ha visto, una de las limitaciones más importantes de los métodos heurísticos es la denominada miopía, provocada por seleccionar, en cada paso, la mejor opción. Resulta muy ilustrativo que en los métodos de inserción se obtengan mejores resultados al elegir al azar el vértice a insertar, que tomar el elemento más cercano. Sin embargo, es evidente que el tomar una opción al azar como norma puede conducirnos a cualquier resultado, por lo que parece más adecuado recurrir a algún procedimiento sistemático que compendie la evaluación con el azar. Se verán dos de los métodos más utilizados.

3.5.1 Procedimientos Aleatorizados

Una modificación en el algoritmo de construcción consiste en sustituir una elección *greedy* por una elección al azar de entre un conjunto de buenos candidatos. Así, en cada paso del procedimiento, se evalúan todos los elementos que pueden ser añadidos y se selecciona un subconjunto con los mejores. La elección se realiza al azar sobre ese subconjunto de buenos candidatos.
Existen varias maneras de establecer el subconjunto de mejores candidatos. En un problema de maximización, en donde cuanto mayor es la evaluación de una opción más "atractiva" resulta, podemos destacar:

- Establecer un número fijo *k* para el subconjunto de mejores candidatos e incluir los *k* mejores.
- Establecer un valor umbral e incluir en el conjunto todos los elementos cuya evaluación esté por encima de dicho valor. Incluir siempre el mejor de todos.
- Establecer un porcentaje respecto del mejor, e incluir en el subconjunto todos aquellos elementos cuya evaluación difiere, de la del mejor en porcentaje, en una cantidad menor o igual que la establecida En todos los casos la elección final se realiza al azar de entre los preseleccionados.

Una estrategia alternativa a la anterior consiste en considerar las evaluaciones como pesos y utilizar un método probabilístico para seleccionar una opción. Así, si $v_1, v_2, ..., v_k$ son los posibles elementos a añadir en un paso del algoritmo, se calculan sus evaluaciones $e_1, e_2, ..., e_k$, y se les asigna un intervalo del siguiente modo:

Elemento	Evaluación	Intervalo
v_1	e_1	$[0, e_1]$
v_2	e_2	$[e_1, e_1 + e_2]$
....	....	....
v_k	e_k	$\left[\sum_{i=1}^{k-1} e_i \sum_{i=1}^{k} e_i\right]$

Tabla 3.1

Se genera un numero *a* al azar entre 0 y $\sum_{i=1}^{k} e_i = 1$, y se selecciona el elemento correspondiente al intervalo que contiene al valor *a*.

Al algoritmo constructivo modificado, tanto con la opción primera como con la segunda, lo llamaremos algoritmo constructivo aleatorizado. Este algoritmo puede que produzca una

solución de peor calidad que la del algoritmo original. Sin embargo, dado que el proceso no es completamente determinista, cada vez que lo realicemos sobre un mismo ejemplo obtendremos resultados diferentes. Esto permite definir un proceso iterativo consistente en ejecutar un numero prefijado de veces (*MAXITER*) el algoritmo y quedarnos con la mejor de las soluciones obtenidas. Obviamente, cada una de dichas soluciones puede mejorarse con un algoritmo de búsqueda local. El siguiente procedimiento incorpora el algoritmo de mejora a dicho esquema.

Algoritmo *de Combinado aleatorizado*

Inicialización
  Obtener una solución con el algoritmo constructivo aleatorizado.
  Sea c^* el costo de dicha solución.
  Hacer $i = 0$
Mientras ($i \prec MAX\_ITER$)
  Obtener una solución $x(i)$ con el algoritmo constructivo aleatorizado
  Aplicar el algoritmo de búsqueda local a $x(i)$.
  Sea $x^*(i)$ la solución obtenida y $S^*(i)$ su valor.
  Si ($S^*(i)$ mejora a c^*)
    Hacer $c^* = S^*(i)$ y guardar la solución actual
  $i = i + 1$

Figura 3.31 Algoritmo de combinado aleatorizado

En cada iteración el algoritmo construye una solución (fase 1) y después trata de mejorarla (fase 2). Así, en la iteración *i*, el algoritmo construye la solución $x(i)$ con valor $S(i)$ y posteriormente la mejora obteniendo $x^*(i)$ con valor $S^*(i)$. Notar que $x^*(i)$ puede ser igual a $x(i)$ si el algoritmo de la segunda fase no encuentra ningún movimiento que mejore la solución.

Después de un determinado número de iteraciones es posible estimar el porcentaje de mejora, obtenido por el algoritmo de la fase 2, y utilizar esta información para aumentar la eficiencia del procedimiento. En concreto, al construir una solución se examina su valor y se puede considerar en la fase 2, la mejoraría en un porcentaje similar al observado en promedio. Si el valor resultante queda alejado del valor de la mejor solución encontrada hasta el momento, podemos descartar la solución actual y no realizar la fase 2 del algoritmo, con el consiguiente ahorro computacional.

Dado el interés por resolver problemas enteros en general, y en particular el PAV, se han propuesto numerosas mejoras y nuevas estrategias sobre el esquema anterior. Una de las más utilizadas es la denominada técnica de Multi-Arranque que se aborda en la siguiente sección.

3.5.2 Métodos Multi-Arranque

Los métodos Multi-Arranque (también llamados Multi-Start o Re-Start) generalizan el esquema anterior. Tienen dos fases: la primera en la que se genera una solución y la segunda en la que la solución es típicamente, pero no necesariamente, mejorada. Cada iteración global produce una solución, usualmente un óptimo local, y la mejor de todas es la salida del algoritmo.

Algoritmo *de Multi-Arranque*

Mientras (Condición de parada)

Fase de Generación
Construir una solución

Fase de Búsqueda
Aplicar un método de búsqueda para mejorar la solución construida

Actualización
Si la solución obtenida mejora a la mejor almacenada, actualizarla.

Figura 3.32 Algoritmo de *Multi-Arranque*

Dada su sencillez de aplicación, estos métodos han sido muy utilizados para resolver gran cantidad de problemas. En el contexto del la programación no lineal sin restricciones, podemos encontrar numerosos trabajos, tanto teóricos como aplicados. [Rinnoy Kan y Timmer, 1989] estudian la generación de soluciones aleatorias (métodos Monte Carlo) y condiciones de convergencia. Las primeras aplicaciones en el ámbito de la optimización combinatoria consistían en métodos sencillos de construcción, completa o parcialmente aleatorios, y su posterior mejora con un método de búsqueda local. Sin embargo, el mismo esquema permite sofisticar el procedimiento, basándolo en unas construcciones y/o mejoras más complejas.

Uno de los artículos que contiene las ideas en que se basa el método Tabú Search [Glover, 1977] también incluye aplicaciones de estas ideas para los métodos de multi-start. Básicamente se trata de almacenar la información relativa a soluciones ya generadas, y utilizarla para la construcción de nuevas soluciones. Las estructuras de memoria reciente y frecuente se introducen en este contexto. Diferentes aplicaciones se pueden encontrar en [Rochat y Taillard, 1995] y [Lokketangen y Glover, 1996].

Utilizando como procedimiento de mejora un algoritmo genético, Ulder y otros, en el año 1990, proponen un método para obtener buenas soluciones al problema del agente viajero. Los autores muestran cómo el uso de las técnicas de re-starting aumenta la eficiencia del algoritmo, comparándolo con otras versiones de heurísticos genéticos sin re-starting.
Un problema abierto actualmente para diseñar un buen procedimiento de búsqueda, basada en multi-arranque es determinar si es preferible implementar un procedimiento de mejora sencillo que permita realizar un gran número de iteraciones globales o, alternativamente, aplicar una rutina más compleja que mejore significativamente unas pocas soluciones generadas. Un procedimiento sencillo depende fuertemente de la solución inicial, pero un método más elaborado consume mucho más tiempo de computación y, por tanto, puede ser aplicado pocas veces, reduciendo el muestreo del espacio de soluciones.
Una de las variantes más populares de estos métodos se denomina GRASP [Feo y Resende, 1995] y está obteniendo resultados excelentes en la resolución de numerosos problemas combinatorios. En la próxima sección se describen a detalle estos métodos.

3.5.3 GRASP (Greedy Randomized Adaptive Search Procedures)

Los métodos GRASP fueron desarrollados al final de la década de los 80 con el objetivo inicial de resolver problemas de cubrimientos de conjuntos [Feo y Resende, 1995] El término GRASP fue introducido por [Feo y Resende, 1995] como una nueva técnica metaheurística de propósito general.
GRASP es un procedimiento de multi-arranque en donde cada paso consiste en una fase de construcción y una de mejora. En la fase de construcción se aplica un procedimiento heurístico constructivo para obtener una buena solución inicial. Esta solución se mejora en la segunda fase mediante un algoritmo de búsqueda local. La mejor de todas las soluciones examinadas se guarda como resultado final.
La palabra GRASP proviene de las siglas de Greedy Randomized Adaptive Search Procedures que en español seria aproximadamente: Procedimientos de búsqueda basados en funciones voraces aleatorizadas que se adaptan. Veamos los elementos de este procedimiento.
En la fase de construcción se produce iterativamente una solución posible, considerando un elemento en cada paso. En cada iteración, la elección del próximo elemento para ser añadido a la solución parcial viene determinada por una función greedy. Esta función mide el beneficio de añadir cada uno de los elementos según la función objetivo y elegir la mejor. Nótese que esta medida es miope en el sentido que no tiene en cuenta qué ocurrirá en iteraciones sucesivas al realizar una elección, sino únicamente en esta iteración.

Se dice que el heurístico greedy se adapta porque en cada iteración se actualizan los beneficios obtenidos al añadir el elemento seleccionado a la solución parcial. Es decir, la evaluación que se tenga de añadir un determinado elemento a la solución en la iteración j, no coincidirá necesariamente con la que se tenga en la iteración $j + 1$.

El heurístico es aleatorizado porque no selecciona el mejor candidato según la función greedy adaptada sino que, con el objeto de diversificar y no repetir soluciones en dos construcciones diferentes, se construye un lista con los mejores candidatos de entre los que se toma uno al azar.

Al igual que ocurre en muchos métodos, las soluciones generadas por la fase de construcción de GRASP no suelen ser óptimos locales. Dado que la fase inicial no garantiza la optimalidad local respecto de la estructura de entorno en la que se esté trabajando (notar que hay selecciones aleatorias), se aplica un procedimiento de búsqueda local como postprocesamiento para mejorar la solución obtenida.
En la fase de mejora se suele emplear un procedimiento de intercambio simple con el objeto de no emplear mucho tiempo en esta mejora. Nótese que GRASP se basa en realizar múltiples iteraciones y quedarse con la mejor, por lo que no es especialmente beneficioso para el método el detenerse demasiado en mejorar una solución dada.
El siguiente esquema muestra el funcionamiento global del algoritmo:

Algoritmo *GRASP*

Mientras (Condición de parada)

Fase Constructiva
Seleccionar una lista de elementos candidatos.
Considerar una Lista restringida de los mejores candidatos.
Seleccionar un elemento aleatoriamente de la lista restringida.

Fase de Mejora
Realizar un proceso de búsqueda local a partir de la solución construida hasta que no se pueda mejorar más.

Actualización
Si la solución obtenida mejora a la mejor almacenada, actualizarla.

Figura 3.33 Algoritmo de *GRASP*

El realizar muchas iteraciones GRASP es una forma de realizar un muestreo del espacio de soluciones. Martí en [Martí, 2000] nos dice que basándose en observaciones empíricas, se observa que la distribución de la muestra generalmente tiene un valor en promedio que es inferior al obtenido por un procedimiento determinista, sin embargo, la mejor de las soluciones encontradas generalmente supera a la del procedimiento determinista con una alta probabilidad.

Las implementaciones GRASP generalmente son robustas en el sentido de que es difícil el encontrar ejemplos patológicos en donde el método funcione arbitrariamente y mal.

Algunas sugerencias que se han encontrado para mejorar el procedimiento son:

Se puede incluir una fase previa a la de construcción: una fase determinista con el objetivo de ahorrar esfuerzo a la fase siguiente.
Si se conoce que ciertas subestructuras forman parte de una solución óptima, éstas pueden ser el punto de partida de la fase constructiva. Tal y como señalan [Feo y Resende,1995] una de las características más relevantes de GRASP es su sencillez y facilidad de implementación. Basta con fijar el tamaño de la lista de candidatos y el número de iteraciones para determinar completamente el procedimiento. De esta forma se pueden concentrar los esfuerzos en diseñar estructuras de datos para optimizar la eficiencia del código y proporcionar una gran rapidez al algoritmo, dado que éste es uno de los objetivos principales del método.

El enorme éxito de este método se puede constatar en la gran cantidad de aplicaciones que han aparecido en los últimos años. [Festa y Resende, 2001] comentan cerca de 200 trabajos en los que se aplica o desarrolla GRASP.

3.6 Búsqueda Tabú

Los orígenes de la Búsqueda Tabú (Tabu Search, TS) pueden situarse en diversos trabajos publicados a finales de los 70 [Glover, 1977]. Oficialmente, el nombre y la metodología fueron introducidos posteriormente por Fred Glover en 1989. Numerosas aplicaciones han aparecido en la literatura, así como artículos y libros para difundir el conocimiento teórico del procedimiento [Glover y Laguna, 1997].

TS es una técnica para resolver problemas combinatorios de gran dificultad que está basada en principios generales de Inteligencia Artificial (IA). En esencia es un metaheurístico que puede ser utilizado para guiar cualquier procedimiento de búsqueda local en la búsqueda agresiva del óptimo del problema. Por agresiva nos referimos a la estrategia de evitar que la búsqueda quede "atrapada" en un óptimo local que no sea global. A tal efecto, TS toma de la IA el concepto de memoria y lo implementa mediante estructuras simples con el objetivo de dirigir la búsqueda teniendo en cuenta la historia de ésta. Es decir, el procedimiento trata de extraer información de lo sucedido y actuar en consecuencia. En este sentido puede decirse que hay un cierto aprendizaje y que la búsqueda es inteligente.

El principio de TS podría resumirse como:

Es mejor una mala decisión basada en información que una buena decisión al azar, ya que, en un sistema que emplea memoria, *una mala elección basada en una estrategia proporcionara claves útiles para continuar la búsqueda. Una buena elección fruto del azar no proporcionará ninguna información para posteriores acciones.* [Martí, 2000]

TS comienza de la misma forma que cualquier procedimiento de búsqueda local, procediendo iterativamente de una solución x a otra y en el entorno de la primera: $N(x)$. Sin embargo, en lugar de considerar todo el entorno de una solución, TS define el entorno reducido $N^*(x)$ como aquellas soluciones disponibles del entorno de x. Así, se considera que a partir de x, sólo las soluciones del entorno reducido son alcanzables.

$$N^*(x) \subseteq N(x)$$

Existen muchas maneras de definir el entorno reducido de una solución. La más sencilla consiste en etiquetar como tabú las soluciones previamente visitadas en un pasado cercano. Esta forma se conoce como memoria a corto plazo (short term memory) y está basada en guardar en una lista tabú T las soluciones visitadas recientemente (Recency).

Así, en una iteración determinada, el entorno reducido de una solución se obtendría como el entorno usual eliminando las soluciones etiquetadas como tabú. $N(x) \setminus T$

El objetivo principal de etiquetar las soluciones visitadas como tabú es el de evitar que la búsqueda se cicle. Por ello se considera que tras un cierto numero de iteraciones la búsqueda está en una región distinta y puede liberarse del status tabú (pertenencia a T) a las soluciones antiguas.
De esta forma se reduce el esfuerzo computacional de calcular el entorno reducido en cada iteración. En los orígenes de TS se sugerían listas de tamaño pequeño, actualmente se considera que las listas pueden ajustarse dinámicamente según la estrategia que se esté utilizando.
Se define un nivel de aspiración como aquellas condiciones que, de satisfacerse, permitirían alcanzar una solución aunque tenga status tabú. Una implementación sencilla consiste en permitir alcanzar una solución siempre que mejore a la mejor almacenada, aunque esté etiquetada tabú. De esta forma se introduce cierta flexibilidad en la búsqueda y se mantiene su carácter agresivo.

Es importante considerar que los métodos basados en búsqueda local requieren de la exploración de un gran número de soluciones en poco tiempo, por ello es crítico el reducir al mínimo el esfuerzo computacional de las operaciones que se realizan a menudo. En ese sentido, la memoria a corto plazo de TS está basada en atributos en lugar de ser explícita; esto es, en lugar de almacenar las soluciones completas (como ocurre en los procedimientos enumerativos de búsqueda exhaustiva) se almacenan únicamente algunas características de éstas.
La memoria mediante atributos produce un efecto más sutil y efectivo en la búsqueda, ya que un atributo o grupo de atributos identifica a un conjunto de soluciones, del mismo modo que los hiperplanos o esquemas utilizados en los algoritmos Genéticos (ver el apartado de algoritmos genéticos). Así, un atributo que fue etiquetado como tabú por pertenecer a una solución visitada hace n iteraciones, puede impedir en la iteración actual, el alcanzar una solución por contenerlo, aunque ésta sea diferente de la que provocó el que el atributo fuese etiquetado. Esto permite, a largo plazo, el que se identifiquen y mantengan aquellos atributos que inducen una cierta estructura beneficiosa en las soluciones visitadas.
Con los elementos descritos puede diseñarse un algoritmo básico de TS para un problema de optimización dado. Sin embargo, TS ofrece muchos más elementos para construir algoritmos realmente potentes y eficaces. A menudo, dichos elementos han sido ignorados en muchas aplicaciones y actualmente la introducción de éstos en la comunidad científica constituye un reto para los investigadores del área.

Un algoritmo TS está basado en la interacción entre la memoria a corto plazo y la memoria a largo plazo. Ambos tipos de memoria llevan asociadas sus propias estrategias y atributos, y actúan en ámbitos diferentes. Como ya se ha mencionado la memoria a corto plazo suele almacenar atributos de soluciones recientemente visitadas, y su objetivo es explorar a fondo una región dada del espacio de soluciones. En ocasiones se utilizan estrategias de listas de candidatos para restringir el número de soluciones examinadas en una iteración dada o para mantener un carácter agresivo en la búsqueda.

La **memoria a largo plazo** almacena las frecuencias u ocurrencias de atributos en las soluciones visitadas tratando de identificar o diferenciar regiones. La memoria a largo plazo tiene dos estrategias asociadas: Intensificar y Diversificar la búsqueda. La intensificación consiste en regresar a regiones ya exploradas para estudiarlas más a fondo. Para ello se favorece la aparición de aquellos atributos asociados a buenas soluciones encontradas.
La diversificación consiste en visitar nuevas áreas no exploradas del espacio de soluciones. Para ello se modifican las reglas de elección para incorporar a las soluciones atributos que no han sido usados frecuentemente. La siguiente tabla 3.2 muestra los elementos mencionados.

Memoria	Atributos	Estrategias	Ámbito
Corto Plazo	Reciente	Tabú-Aspiración Listas de Candidatos	Local
Largo Palzo	Frecuente	Intensif-Diversif	Global

Tabla 3.2

Existen otros elementos más sofisticados dentro de TS que, aunque poco probados, han dado muy buenos resultados en algunos problemas. Entre ellos se pueden destacar:

Movimientos de Influencia: Son aquellos movimientos que producen un cambio importante en la estructura de las soluciones. Usualmente, en un procedimiento de búsqueda local, la búsqueda es dirigida mediante la evaluación de la función objetivo. Sin embargo, puede ser muy útil el encontrar o diseñar otros evaluadores que guíen a ésta en determinadas ocasiones.

Los movimientos de influencia proporcionan una evaluación alternativa de la bondad de los movimientos al margen de la función objetivo. Su utilidad principal es la determinación de estructuras subyacentes en las soluciones. Esto permite que sean la base para procesos de Intensificación y Diversificación a largo plazo.

Oscilación Estratégica: La oscilación estratégica opera orientando los movimientos en relación a una cierta frontera en donde el método se detendría normalmente. Sin embargo, en vez de detenerse, las reglas para la elección de los movimientos se modifican para permitir que la región al otro lado de la frontera sea alcanzada. Posteriormente se fuerza al procedimiento a regresar a la zona inicial. El proceso de aproximarse, traspasar y volver sobre una determinada frontera crea un patrón de oscilación que da nombre a esta técnica. Una implementación sencilla consiste en considerar la barrera de la factibilidad \ infactibilidad de un problema dado. Implementaciones más complejas pueden crearse identificando determinadas estructuras de soluciones que no son visitadas por el algoritmo y considerando procesos de construcción / destrucción asociados a éstas. La oscilación estratégica proporciona un medio adicional para lograr una interacción muy efectiva entre intensificación y diversificación.

Elecciones Probabilísticas: Normalmente TS se basa en reglas sistemáticas en lugar de decisiones al azar. Sin embargo, en ocasiones se recomienda el aleatorizar algunos procesos para facilitar la elección de buenos candidatos o cuando no está clara la estrategia a seguir (quizá por tener criterios de selección enfrentados). La selección aleatoria puede ser uniforme o seguir una distribución de probabilidad construida empíricamente a partir de la evaluación asociada a cada movimiento.

Umbrales Tabú: El procedimiento conocido como Tabú Thresholding (TT) se propone para aunar ideas que provienen de la oscilación estratégica y de las estrategias de listas de candidatos en un marco sencillo que facilite su implementación. El uso de la memoria es implícito en el sentido que no hay una lista tabú en donde anotar el status de los movimientos, pero la estrategia de elección de los mismos previene el ciclado. TT utiliza elecciones probabilísticas y umbrales en las listas de candidatos para implementar los principios de TS.

Re-encadenamiento de Trayectorias (Path Relinking): Este método se basa en volver a unir dos buenas soluciones mediante un nuevo camino. Así, si en el proceso de búsqueda hemos encontrado dos soluciones x e y con un buen valor de la función objetivo, podemos considerar el tomar x como solución inicial e y como solución final e iniciar un nuevo camino desde x hasta y. Para seleccionar los movimientos no consideraremos la función objetivo o el criterio que hayamos estado utilizando hasta el momento, sino que iremos incorporando a x los atributos de y hasta llegar a ésta. Por eso esperamos que alguna de las soluciones intermedias que se visitan en este proceso de "entorno constructivo" sea muy buena. En algunas implementaciones se ha considerado el explorar el entorno de las soluciones intermedias para dar más posibilidad al descubrimiento de buenas soluciones. Detalles sobre el método pueden encontrarse en [Glover, Laguna y Martí, 2000].

[Laguna y Martí, 1999] proponen el uso de Path Relinking (PR) en el contexto de GRASP, aunque aquí su significado es diferente ya que las soluciones no han estado unidas por ningún camino previo. Así, una vez generada una colección de soluciones mediante las fases de construcción y mejora de GRASP, se seleccionan parejas (o subconjuntos) de soluciones para unirlas mediante PR. A partir de una solución se realiza una búsqueda local para llegar a la otra (o a una combinación de las otras en el caso de subconjuntos).
Es importante destacar el hecho de que muchas de las aplicaciones basadas en TS no utilizan los últimos elementos descritos, por lo que son susceptibles de ser mejoradas. Al mismo tiempo, los éxitos de las numerosas implementaciones del procedimiento han llevado la investigación hacia formas de explotar con mayor intensidad sus ideas subyacentes. En este campo podemos destacar los últimos trabajos de *Modelos de entrenamiento y aprendizaje tabú, Maquinas tabú y Diseño tabú.*

3.7 Recocido Simulado ó templado simulado (Simulated Annealing SA)

Kirpatrick, Gelatt y Vecchi proponen en 1983 un procedimiento para obtener soluciones aproximadas a problemas de optimización, llamado Simulated Annealing (Templado simulado). Este procedimiento se basa en una analogía con el comportamiento de un sistema físico al someterlo a un baño de agua caliente. Simulated Annealing (SA) ha sido probado con éxito en numerosos problemas de optimización, mostrando gran "habilidad" para evitar quedar atrapado en óptimos locales. Debido a su sencillez de implementación así como a los buenos resultados que iban apareciendo, experimentó un gran auge en la década de los 80. Kirpatrick y otros trabajando en el diseño de circuitos electrónicos consideraron aplicar el algoritmo de Metrópolis en alguno de los problemas de optimización combinatoria que aparecen en este tipo de diseños. El algoritmo de Metrópolis simula el cambio de energía en el proceso de enfriamiento de un sistema físico. Las leyes de la termodinámica establecen que, a una temperatura t, la probabilidad de un aumento de energía de magnitud ∂E viene dada por la expresión siguiente:

$$P(\partial E) = e^{\frac{-\partial E}{kt}}$$

donde *k* es la constante de Boltzmann.

La simulación de Metrópolis genera una perturbación y calcula el cambio resultante en la energía. Si ésta decrece, el sistema se mueve al nuevo estado, en caso contrario, se acepta el nuevo estado de acuerdo con la probabilidad dada en la ecuación anterior.
Kirpatrick y otros, pensaron que era posible establecer una analogía entre los parámetros que intervienen en la simulación termodinámica de Metrópolis y los que aparecen en los métodos de optimización local, tal y como muestra la tabla adjunta.

Termodinámica	Optimización
Configuración	Solución posible
Configuración Fundamental	Solución posible
Estrategia de la Configuración	Costo de la solución

Tabla 3.3

Para ello, establecen un paralelismo entre el proceso de las moléculas de una sustancia que van colocándose en los diferentes niveles energéticos buscando un equilibrio, y las soluciones visitadas por un procedimiento de búsqueda local. Así pues, SA es un

procedimiento basado en búsqueda local en donde todo movimiento de mejora es aceptado y se permiten movimientos de *no mejora,* de acuerdo con unas probabilidades.

Dichas probabilidades están basadas en la analogía con el proceso físico de enfriamiento y se obtienen como función de la temperatura del sistema. La estrategia de SA es comenzar con una temperatura inicial *alta*, lo cual proporciona una probabilidad también alta de aceptar un movimiento de *no mejora*. En cada iteración se va reduciendo la temperatura y por lo tanto las probabilidades son cada vez más pequeñas conforme avanza el procedimiento y nos acercamos a la solución óptima. De este modo, inicialmente se realiza una diversificación de la búsqueda sin controlar demasiado el costo de las soluciones visitadas. En iteraciones posteriores resulta cada vez más difícil el aceptar malos movimientos, y por lo tanto se produce un descenso en el costo.
De esta forma, SA tiene la habilidad de salir de óptimos locales al aceptar movimientos de *no mejora* en los estados intermedios. Al final del proceso éstos son tan poco probables que no se producen, por lo que, si no hay movimientos de mejora, el algoritmo finaliza. El diagrama siguiente muestra un esquema general del algoritmo para un problema de minimización:

Algoritmo *Recocido Simulado*

Tomar una solución inicial x
Tomar una temperatura inicial T

Mientras (no congelado)
    Realizar L veces
        Tomar x' de $N(x)$.
        Si $(d \prec 0)$ hacer $x = x'$
        Si $(d \succ 0)$ hacer $x = x'$ con $p = d^{-\frac{e}{T}}$
        Hacer $T = rT$

Figura 3.34 Algoritmo *Recocido Simulado*

Para diseñar un algoritmo a partir del esquema general anterior hay que determinar los parámetros del sistema:

- **La temperatura inicial** se suele determinar, realizando una serie de pruebas para alcanzar una determinada fracción de movimientos aceptados.
- **La velocidad de enfriamiento** r.
- La longitud L se toma habitualmente proporcional al tamaño esperado de $N(x)$.
- El criterio para finalizar la **secuencia de enfriamiento** (estado de congelación). Usualmente hacemos $cont = cont + 1$ cada vez que se completa una temperatura y el porcentaje de movimientos aceptados es menor de la cantidad *MinPercent*. Contrariamente hacemos $cont = 0$ cuando se mejora la mejor solución almacenada.

La clave de una implementación de SA es el manejo de la cola o secuencia de enfriamiento: Cuál es la temperatura inicial y cómo disminuye la temperatura en cada iteración, son las dos preguntas a responder para diseñar un algoritmo. Podemos encontrar numerosos estudios en donde se comparan colas lentas (aquellas en las que la temperatura disminuye poco a poco) con colas rápidas que vienen a ser métodos de descenso simple con pequeñas perturbaciones.

En [Johnson y otros, 1989] se sugiere el realizar mejoras y especializaciones aunque no se basen en la analogía física. Además, aconsejan utilizar soluciones posibles y no posibles cuando la estructura del problema lo permita, penalizando en la función de evaluación las soluciones imposibles. El objetivo es diversificar y considerar más soluciones (Notar que con bajas temperaturas la solución tendera a ser posible).
Las últimas implementaciones de SA apuntan a olvidarse de la analogía física y manejar la cola de enfriamiento como una estructura de memoria. Es decir, la probabilidad de aceptar o rechazar un movimiento de no mejora depende no de la iteración (tiempo transcurrido) sino de lo sucedido en la búsqueda. En este sentido, la probabilidad será función de algunas variables de estado del proceso. En la actualidad se están diseñando numerosos algoritmos híbridos en donde la búsqueda local se realiza con un procedimiento basado en SA, en ocasiones combinado con búsqueda Tabú.

3.8 Métodos Evolutivos

Los métodos evolutivos están basados en poblaciones de soluciones. A diferencia de los métodos vistos hasta ahora, en cada iteración del algoritmo no se tiene una única solución sino un conjunto de éstas. Estos métodos se basan en generar, seleccionar, combinar y reemplazar un conjunto de soluciones. Dado que mantienen y manipulan un conjunto en lugar de una solución a lo largo de todo el proceso de búsqueda, suelen presentar tiempos de computación sensiblemente más altos que los de otros metaheurísticos. Este hecho se ve agravado por la convergencia de la población que requiere de un gran numero de iteraciones en el caso de los algoritmos genéticos, tal y como se describe a continuación.

3.8.1 Algoritmos Genéticos

Los Algoritmos Genéticos (GA) fueron introducidos por John Holland en 1970 inspirándose en el proceso observado en la evolución natural de los seres vivos.
Los biólogos han estudiado en profundidad los mecanismos de la evolución, y aunque quedan parcelas por entender, muchos aspectos están bastante explicados. De manera muy general se puede decir que en la evolución de los seres vivos el problema al que cada individuo se enfrenta cada día es la supervivencia. Para ello cuenta con las habilidades innatas provistas en su material genético. A nivel de los genes, el problema es el de buscar aquellas adaptaciones beneficiosas en un medio hostil y cambiante. Debido en parte a la selección natural, cada especie gana una cierta cantidad de "conocimiento", el cual es incorporado a la información de sus cromosomas.
Así pues, la evolución tiene lugar en los cromosomas, en donde está codificada la información del ser vivo. La información almacenada en el cromosoma varía de unas generaciones a otras. En el proceso de formación de un nuevo individuo, se combina la información cromosómica de los progenitores aunque la forma exacta en que se realiza es aún desconocida. Aunque muchos aspectos están todavía por discernir, existen unos principios generales de la evolución biológica ampliamente aceptados por la comunidad científica. Algunos de éstos son:

- La evolución opera en los cromosomas en lugar de en los individuos a los que representan.
- La selección natural es el proceso por el que los cromosomas con "buenas estructuras" se reproducen más a menudo que los demás.
- En el proceso de reproducción tiene lugar la evolución mediante la combinación de los cromosomas de los progenitores. Llamamos Recombinación a este proceso en el

que se forma el cromosoma del descendiente. También son de tener en cuenta las mutaciones que pueden alterar dichos códigos.
- La evolución biológica no tiene memoria en el sentido de que en la formación de los cromosomas únicamente se considera la información del periodo anterior.

Los algoritmos genéticos establecen una analogía entre el conjunto de soluciones de un problema y el conjunto de individuos de una población natural, codificando la información de cada solución en un string (vector binario) a modo de cromosoma. En palabras del propio Holland: *Se pueden encontrar soluciones aproximadas a problemas de gran complejidad computacional mediante un proceso de evolución simulada.* [Martí, 2000]

Para tal efecto se introduce una función de evaluación de los cromosomas, que llamaremos calidad ("fitness") basada en la función objetivo del problema. Igualmente, se introduce un mecanismo de selección de manera que los cromosomas con mejor evaluación sean escogidos para "reproducirse" más frecuentemente que los que la tienen peor.
Los algoritmos desarrollados por Holland en 1992, inicialmente eran sencillos, pero dieron buenos resultados en problemas considerados difíciles. Un primer esquema de un algoritmo genético se muestra en la figura 3.35.

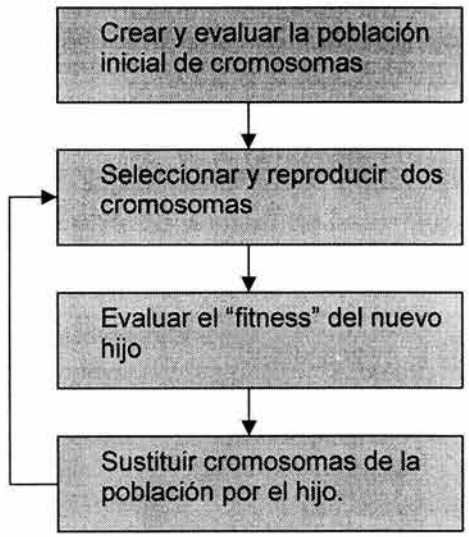


Figura 3.35 Esquema de un algoritmo genético

Los algoritmos genéticos están basados en integrar e implementar eficientemente dos ideas fundamentales: Las representaciones simples como strings binarios de las soluciones del problema y la realización de transformaciones simples para modificar y mejorar estas representaciones.

Para llevar a la práctica el esquema anterior y concretarlo en un algoritmo, hay que especificar los siguientes elementos, que se comentarán a continuación:

- Una representación cromosómica.
- Una población inicial.
- Una medida de evaluación.
- Un criterio de selección / eliminación de cromosomas.

- Una o varias operaciones de recombinación.
- Una o varias operaciones de mutación.

En los trabajos originales las soluciones se representaban por strings binarios, es decir, listas de 1s y 0s. Este tipo de representaciones ha sido ampliamente utilizada incluso en problemas en donde no es muy natural. En 1985, De Jong introduce la siguiente cuestión: ¿Qué se debe hacer cuando los elementos del espacio de búsqueda se representan de modo natural por estructuras complejas como vectores, árboles o grafos?, ¿Se debe intentar linealizar en un string o trabajar directamente con estas estructuras?

En la actualidad podemos distinguir dos escuelas, la primera se limita a strings binarios, mientras que la segunda utiliza todo tipo de configuraciones. Hemos de notar que las operaciones genéticas dependen del tipo de representación, por lo que la elección de una condiciona a la otra. La ventaja de las primeras es que permite definir fácilmente operaciones de recombinación, además los resultados sobre convergencia están probados para el caso de strings binarios. Sin embargo en algunos problemas puede ser poco natural y eficiente el utilizarlas. Por ejemplo en el problema del agente viajero sobre 5 ciudades y 20 aristas, el string 01000100001000100010 representa una solución sobre las aristas ordenadas.
Sin embargo, dicha representación no es muy natural y, además, no todos los strings con cinco 1s representan soluciones, lo cual complica substancialmente la definición de una operación de sobrecruzamiento. Es más natural la ruta de ciudades: $(2,3,1,5,4)$, lo cual permite definir naturalmente diferentes operaciones estables.

La población inicial suele ser generada aleatoriamente. Sin embargo, últimamente se están utilizando métodos heurísticos para generar soluciones iniciales de buena calidad. En este caso, es importante garantizar la diversidad estructural de estas soluciones para tener una representación de la mayor parte de población posible o al menos evitar la convergencia prematura. Respecto de la evaluación de los cromosomas, se suele utilizar la calidad como medida de la bondad según el valor de la función objetivo en el que se puede añadir un factor de penalización para controlar la infactibilidad.
Este factor puede ser estático o ajustarse dinámicamente, lo cual produciría un efecto similar al de la oscilación estratégica en Tabu Search:

$$\begin{aligned} calidad \;=\; & Valor\ Objetivo\ Normalizado \\ & - Penalizacion\ X\ Medida\ Infactibilidad \end{aligned}$$

La selección de los padres viene dada, habitualmente, mediante probabilidades según su fitness. Uno de los procedimientos más utilizado es el denominado de la ruleta en donde cada individuo tiene una sección circular de una ruleta que es directamente proporcional a su calidad. Para realizar una selección se realizaría una tirada de bola en la ruleta, tomando el individuo asociado a la casilla donde cayó la bola.
Los Operadores de Cruzamiento más utilizados son:

- **De un punto**: Se elige aleatoriamente un punto de ruptura en los padres y se intercambian sus bits.
- **De dos puntos**: Se eligen dos puntos de ruptura al azar para intercambiar.
- **Uniforme**: En cada bit se elige al azar un padre para que contribuya con su bit al del hijo, mientras que el segundo hijo recibe el bit del otro padre.

- **PMX, SEX**: Son operadores más sofisticados fruto de mezclar y aleatorizar los anteriores.

La operación de **Mutación** más sencilla, y una de las más utilizadas consiste en reemplazar con cierta probabilidad el valor de un bit. Notar que el papel que juega la mutación es el de introducir un factor de diversificación, ya que, en ocasiones, la convergencia del procedimiento a buenas soluciones puede ser prematura y quedarse atrapado en óptimos locales. Otra forma obvia de introducir nuevos elementos en una población es recombinar elementos tomados al azar, sin considerar su fitness.

A continuación se muestra la implementación de un algoritmo genético propuesta por [Michalewicz, 1996], en donde se combinan los elementos genéticos con la búsqueda local.

Algoritmo *Genético*

1. **Generar soluciones** — Construir un conjunto de soluciones P con tamaño PopSize mediante generación aleatoria.
2. **Mejorar soluciones** — Aplicar un método de búsqueda local a cada solución del conjunto P.

Mientras (número de evaluaciones < *MaxEval*)

3. **Evaluación** — Evaluar las soluciones en P y actualizar, si es necesario, la major solución almacenada.
4. **Supervivencia** — Calcular la probabilidad de supervivencia basada en la calidad de las soluciones. Según dichas probabilidades seleccionar aleatoriamente PopSize soluciones (con reemplazamiento) de P. Sea P el nuevo conjunto formado por las soluciones seleccionadas (algunas pueden aparecer repetidas).
5. **Combinación** — Seleccionar una fracción pc de soluciones de P para ser combinadas. La selección es aleatoria y equi-probable para todos los elementos de P. Los elementos seleccionados se emparejan al azar y, por cada pareja, se generan dos descendientes que reemplazarán a los padres en P.
6. **Mutación** — Una fracción pm de las soluciones de P se selecciona para aplicar el operador de mutación. La solución resultante reemplaza a la original en P.

Figura 3.36 Algoritmo *Genético*

Convergencia del Algoritmo

Dado que el algoritmo genético opera con una población en cada iteración, se espera que el método converja de modo que al final del proceso la población sea muy similar, y en el infinito se reduzca a un solo individuo. Se ha desarrollado toda una teoría para estudiar la convergencia de estos algoritmos en el caso de strings binarios. Esta teoría se basa principalmente en considerar que un strings un representante de una clase de equivalencia o esquema, reinterpretando la búsqueda en lugar de entre strings, entre esquemas. De este modo se concluye lo que se conoce como **paralelismo intrínseco**:

En una población de m strings se están procesando implícitamente $O(m^3)$ *esquemas.*

A partir de este resultado el teorema de esquemas prueba que la población converge con unos esquemas que cada vez son más parecidos, y en el límite a un único string. En el caso de strings no binarios se introducen los conceptos de *forma* y *conjunto de similitud* que generalizan al de esquema. Se consideran una serie de condiciones sobre los operadores de manera que se garantice la convergencia. Básicamente se exige que al cruzar dos strings de

la misma clase se obtenga otro dentro de ésta. Además hay que respetar ciertas condiciones sobre selección de los progenitores. Bajo toda esta serie de hipótesis se prueba la convergencia del algoritmo. En la práctica no se suelen respetar las condiciones vistas, ya que son difíciles de seguir y probar, encontrándonos que, en ocasiones los algoritmos genéticos resuelven satisfactoriamente un problema de optimización dado, y en otras se quedan muy alejados del óptimo. Los estudiosos del tema han tratado de caracterizar lo que han denominado problemas *AG-fáciles* (Aquellos en los que los AG proporcionan buenos resultados) y *AG-difíciles,* con el objetivo de saber de antemano, al estudiar un nuevo problema, si los AG son una buena elección para su resolución.

Se han tratado de caracterizar estas clases mediante el concepto de *engaño,* considerando que si el algoritmo converge al mejor esquema (aquel con mejor promedio del fitness de sus strings) y en éste se encuentra el óptimo, entonces es fácil que se resuelva satisfactoriamente.
En caso de que el óptimo esté en un esquema con bajo promedio se denomina engaño y se pensaba que en estos casos es cuando el problema es AG-difícil. Sin embargo se ha visto que esta caracterización mediante el engaño no es siempre cierta y no constituye un criterio fiable.
Es importante citar que, a diferencia de otros metaheurísticos, los Algoritmos Genéticos han crecido de forma espectacular, hasta el punto de encontrar referencias sobre ellos, en revistas de informática de carácter general. Además, muchos de los investigadores de este campo están trabajando en desarrollar los aspectos teóricos de la materia, e incorporando algunas otras técnicas de búsqueda local en el esquema genético. Los Algoritmos Meméticos son un caso particular de estos nuevos híbridos que aúnan los elementos de combinación con los de búsqueda local, bajo el nombre de cooperación y competición. Podemos encontrar esquemas de procedimientos genéticos ya implementados en los que incorporar nuestras funciones de evaluación y con poco más, tener un algoritmo genético para nuestro problema. Existen numerosos accesibles en internet (algunos de libre distribución y otros comercializados por compañías de software) y son muy populares, especialmente en entornos informáticos. Estos esquemas reciben el nombre de *context independent* y habitualmente sólo utilizan la evaluación de la solución como información del problema. En general proporcionan resultados de menor calidad que la de los algoritmos específicos (dependientes del contexto) diseñado para un problema dado, pero por otro lado, su aplicación es muy sencilla.

3.9 Búsqueda dispersa

La Búsqueda Dispersa (BD, en inglés Scatter Search) es un método evolutivo que ha sido aplicado en la resolución de un gran número de problemas de optimización. Los conceptos y principios fundamentales del método, fueron propuestos a comienzo de la década de los setenta, basados en las estrategias para combinar reglas de decisión, especialmente en problemas de secuenciación. La BD se basa en el principio de que la información sobre la calidad o el atractivo de un conjunto de reglas, restricciones o soluciones puede ser utilizado mediante la combinación de éstas. En concreto, dadas dos soluciones, se puede obtener una nueva mediante su combinación de modo que mejore a las que la originaron. Al igual que los algoritmos genéticos, el método que nos ocupa se basa en mantener un conjunto de soluciones y realizar combinaciones con éstas; pero a diferencia de éstos no está fundamentado en la aleatorización sobre un conjunto relativamente grande de soluciones sino en las elecciones sistemáticas y estratégicas sobre un conjunto pequeño de éstas. Como ilustración basta decir que los algoritmos genéticos suelen considerar una población

de 100 soluciones mientras que en la búsqueda dispersa es habitual trabajar con un conjunto de tan sólo 10 soluciones.
La primera descripción del método fue publicada en 1977 por Fred Glover donde establece los principios de la BD. En este primer artículo se determina que la BD realiza una exploración sistemática sobre una serie de buenas soluciones llamadas conjunto de referencia. Los siguientes aspectos resumen los principales aspectos de este trabajo:[Martí, 2001]

- El método se centra en combinar dos o más soluciones del conjunto de referencia. La combinación de más de dos soluciones tiene como objetivo el generar centroides.
- Generar soluciones en la línea que unen dos dadas se considera una forma reducida del método.
- Al combinar se deben de seleccionar pesos apropiados y no tomar valores al azar.
- Se deben de realizar combinaciones convexas "no convexas" de las soluciones.
- La distribución de los puntos se considera importante y deben de tomarse dispersos.

La búsqueda dispersa es un método relativamente reciente que hemos de considerar en desarrollo. Durante los últimos años se han realizado nuevas contribuciones aplicando esta metodología en la resolución de conocidos problemas de optimización. Algunas de estas aplicaciones han abierto nuevos campos de estudio, ofreciendo alternativas a los diseños conocidos. [Laguna y Martí, 2002] realizan una revisión de los aspectos clave del método desarrollados durante estos años.
Como conclusión se puede decir que la búsqueda dispersa es un método evolutivo que se encuentra en desarrollo. Sus orígenes se pueden situar en la década de los 70 y, aunque menos conocida que los algoritmos genéticos, se está aplicando en la resolución de numerosos problemas difíciles de optimización.

En la actualidad no existe un esquema único para aplicar la búsqueda dispersa.

Capítulo 4. Caso práctico del Problema del Agente Viajero

En este capítulo realizo una metodología para ayudar a resolver el problema del agente viajero. Así, como el software que se ha desarrollado para resolver este problema y expongo unos estudios comparativos que abordan la eficiencia de los algoritmos heurísticos y exactos que se han realizado. Con la finalidad de tener una idea de qué algoritmos se aplican y a qué software remitirse, al mismo tiempo actualizo la información referente al desarrollo de nuevos algoritmos en su mayoría heurísticos, y la complejidad asociada y el número de nodos que soportan. También expongo algunas aplicaciones que toman como base este problema.

Introducción

Como se ha venido mencionando el problema del agente viajero es un problema extremadamente estudiado, para muestra basta con ver la gran cantidad de artículos que hablan de este problema, dado que es muy fácil de enunciarlo o definirlo, pero resolverlo es muy difícil. Ya lo afirma la teoría de la complejidad computacional en su versión como problema de decisión: es un problema NP-Completo o bien un problema NP-Hard en su versión como un problema de optimización.
También es un problema que llama la atención porque las aplicaciones que tiene sobre todo en planificación y secuenciación de tareas en máquinas y además sirve como base para resolver otros problemas.
También es un problema que ha sido utilizado como "conejillo de indias", en cuanto la aplicación de los algoritmos que se van desarrollando, dado que como se mencionaba es un problema NP-Completo, y si se logra descubrir un algoritmo eficiente y se prueba en el agente viajero querrá decir que el problema del agente viajero pertenece a la clase P. Con lo cual se demostraría que P=NP. Lo cual ha traído de cabeza a los investigadores y diseñadores de algoritmos.

4.1 Metodología de solución del Problema del Agente Viajero PAV

Es importante señalar, que el problema del agente viajero, consiste en encontrar la ruta mínima entre un conjunto de posibilidades de rutas. Sujeto a que debe regresar a la ciudad de la cual partió y no visitar una ciudad más de una ocasión. Antes de seguir la metodología propuesta se debe contar con los datos de entrada del problema: el número de ciudades (nodos) y el costo que representa viajar de una ciudad a otra. El costo puede estar en función del costo de gasolina, tiempo, o bien otras variables.

Cuando se trata de resolver una instancia del PAV, nos daremos cuenta que nos encontramos con varias dificultades. Puede ocurrir que no contemos con un algoritmo para resolver el PAV de manera óptima y que si contamos con éste, el tiempo o el conocimiento no nos permitan el desarrollo o la implementación de tal algoritmo. O bien el PAV puede ser muy grande y aun cuando contemos con el mejor de los algoritmos para intentar encontrar la solución óptima no puede ser posible por el gran costo que implica en tiempo, y tal vez en memoria, computacionalmente hablando.

Ahora bien, como se trata de resolver un problema de optimización y por consiguiente encontrar la mejor solución u óptima de entre un conjunto de alternativas. Para resolver grandes problemas de este tipo podemos elegir entre dos caminos. Si se quiere obtener la respuesta de manera rápida, conformándose con una solución cercana al óptimo o bien obtener la respuesta óptima, pero arriesgándose a obtener la respuesta en un tiempo de computadora impracticable. Entre los que siguen la primera opción destacan los algoritmos

de aproximación o bien algoritmos heurísticos; y la segunda opción da lugar a los *métodos de enumeración* y las técnicas de *programación dinámica*, por mencionar algunos, ya que existen diversos métodos exactos como planos de corte y rama y corte.

Para empezar a resolver el PAV es necesario tener presente que sus soluciones admiten una doble interpretación: mediante grafos y mediante permutaciones, dos herramientas de representación muy habituales en problemas combinatorios. Así, si se plantea el agente viajero como un grafo o bien como un problema de programación matemática finalmente tendremos como resultado o un circuito hamiltoniano o bien la permutación óptima. Se hace necesario plantear el problema del agente viajero a partir de matrices de adyacencia, ya que, si se observa, los algoritmos de aproximación exacta como el de ramificación y acotamiento, se necesita que el problema parta de una matriz de adyacencia. Como las matrices de adyacencia son relativamente sencillas de comprender se sugiere que se aterrice el problema como una matriz de adyacencia. Los números que tienen que figurar en ésta matriz, tienen que ser positivos y enteros. Si el problema se plantea en un plano el problema se le denomina problema del agente viajero geométrico, conocido como el Geometric Traveling Salesman Problem (GTS o PAVG), en donde se tienen ciudades en el plano y se quiere encontrar si hay un circuito solución del PAV que no se pase de, por ejemplo, 1000 Km. La distancia es la euclidiana en términos de enteros (redondeo hacia arriba). En este caso las distancias se consideran distancias euclidianas discretizadas es decir:

$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)}$.

Se comentaba en el capítulo 1 que tan solo un problema de 25 ciudades (que es un problema pequeño para muchos casos prácticos) y una computadora que pudiera ser programada para examinar soluciones a razón de un billón de soluciones por segundo; la computadora terminaría su tarea, en alrededor de 19,674 años.

¡No tiene sentido resolver de esa forma un problema, si al interesado no le alcanza su vida para ver la respuesta!

Ahora bien el tomador de decisiones, además de decidir si quiere saber la respuesta de manera rápida o no, el decidor, también tendrá que tener presente que los algoritmos que resuelven el problema de manera exacta según la literatura, un problema mayor de 30 (que es un problema poco práctico, dado que los problemas reales superan en gran medida este número nodos) es un problema que no se resuelve de manera rápida, cuando uno quiere resolverlo a mano (a menos que se cuente con meses de tiempo para resolver el problema), es decir, tendrá que ser necesaria una computadora y un software que nos ayude a resolver nuestro problema.

Es por esto que se recomienda usar cualquiera de los dos tipos de algoritmos, programados en una computadora. Ahora bien, si tienes una instancia de 10 nodos puedes resolverlo a mano y darte el lujo de obtener la solución óptima mediante programación dinámica o por otro algoritmo de programación exacta, como ramificación y acotamiento.

Pero si se tiene una instancia de 400 nodos es necesario contar con una computadora que soporte un software capaz de analizar soluciones en un tiempo razonable, por ejemplo una de las características principales con las que cuentan las computadoras que resolvieron el tamaño de instancia más grande, hasta el momento, es una Compaq EV6 Alpha con un procesador de 500 MHz. Instalada en paralelo, lo que nos hace pensar que una

computadora con características inferiores será suficiente para ayudarnos a resolver nuestro problema.

En el caso de la tesis de [Castañeda, 2000], todos los métodos implementados fueron programados en Java en una Power PC, Macintosh 6500/225, y resuelve instancias de hasta 100 ciudades y el problema del agente viajero puede ser: simétrico, asimétrico y geométrico. Y los algoritmos que se utilizan son: heurísticos y exactos.
Bajo las premisas anteriores podemos determinar una cierta metodología que se puede apreciar de manera muy general para resolver el problema del agente viajero.

4.1.1 Determinar el número de agentes viajeros

En este paso tenemos que ver con cuantos agentes viajeros cuenta nuestro problema ya que el problema clásico es el que cuenta con sólo un agente viajero, que conocemos como PAV o TSP, pero podemos contar con múltiples agentes viajeros, el cual, recibiría un tratamiento diferente. En [Jianyu,1985], se expone de manera muy clara los métodos de solución de éste problema, y se observa que, cuando contamos con un problema con múltiples agentes viajeros estamos hablando del problema del "*Vehicule Routing Problem*" o VRP. Los siguientes pasos de la metodología que se muestra a continuación, está enfocada cuando se tiene únicamente un agente viajero. Es decir, estamos hablando del problema del agente viajero clásico.

4.1.2 Determinar el tipo de agente viajero

Una vez que conocemos el número de agentes viajeros es necesario saber la estructura que tiene nuestro problema. Los problemas pueden variar por las características con las que cuente nuestro problema, pero finalmente si se aterriza mediante una matriz de adyacencia será más que suficiente. Es decir, tenemos que contar con los costos asociados de viajar de la ciudad i a la ciudad j, o bien, el tiempo o cualquier otro parámetro que quieras medir. Claro, nos podemos apoyar mediante la elaboración de un grafo y recordar que la solución admite doble interpretación o bien un circuito hamiltoniano o una permutación.

Como se comentaba anteriormente el PAV puede ser geométrico o euclidiano y ambos pueden ser simétricos o asimétricos. En un sentido más amplio al PAV que es euclidiano y que además cumple con la desigualdad del triángulo se le llama PAV Geométrico. Así, el PAV puede ser Geométrico, Euclidiano, Simétrico y Asimétrico.

Es necesario saber qué restricciones adicionales tendrá nuestro problema, por ejemplo si se quiere que parta de una ciudad en especial o bien que el agente viajero salga a determinada hora de un nodo y regrese en otro tiempo determinado, o que siga una secuencia previa.

De acuerdo con los datos con los que se cuenten, el PAV puede ser fácilmente transformado en otros problemas; por ejemplo, como un problema de asignación. Así pues, el PAV tiene muchas variantes: con ventanas de tiempo o dependencia horaria, con múltiples agente viajeros, ya que se le pueden agregar características muy específicas al agente viajero. A continuación se muestra un esquema Figura 4.1.a) en donde se puede observar algunas transformaciones del agente viajero. Aquí nos podemos dar cuenta que hay tantos problemas como formas de plantearlo, lo importante es determinar exactamente la estructura del problema. Otro punto importante es como se van a obtener los datos, si tienes costos, distancias o tiempo. De hecho, los costos pueden estar en función del tiempo y la distancia de viajar de una ciudad a otra. Puedes tener datos como latitud y altitud de cada ciudad y

transformarlos para obtener un matriz de adyacencia con números enteros positivos, por ejemplo. Así es como se encuentran los datos en los ejemplos de TSPLIB. Que es un formato que conocen los académicos. Se recomienda visitar la página electrónica www.math.princeton.edu/tsp/usa) para tener más detalle de estos datos. O bien es el tipo de datos que ocupan los programas denominados GIS (Sistemas de información geográfica) más adelante se verá con mayor detalle este tipo de software, dado que para resolver el ejemplo que se propone se utilizó ARCVIEW en la versión 3.2, que es un software que contiene información geográfica.

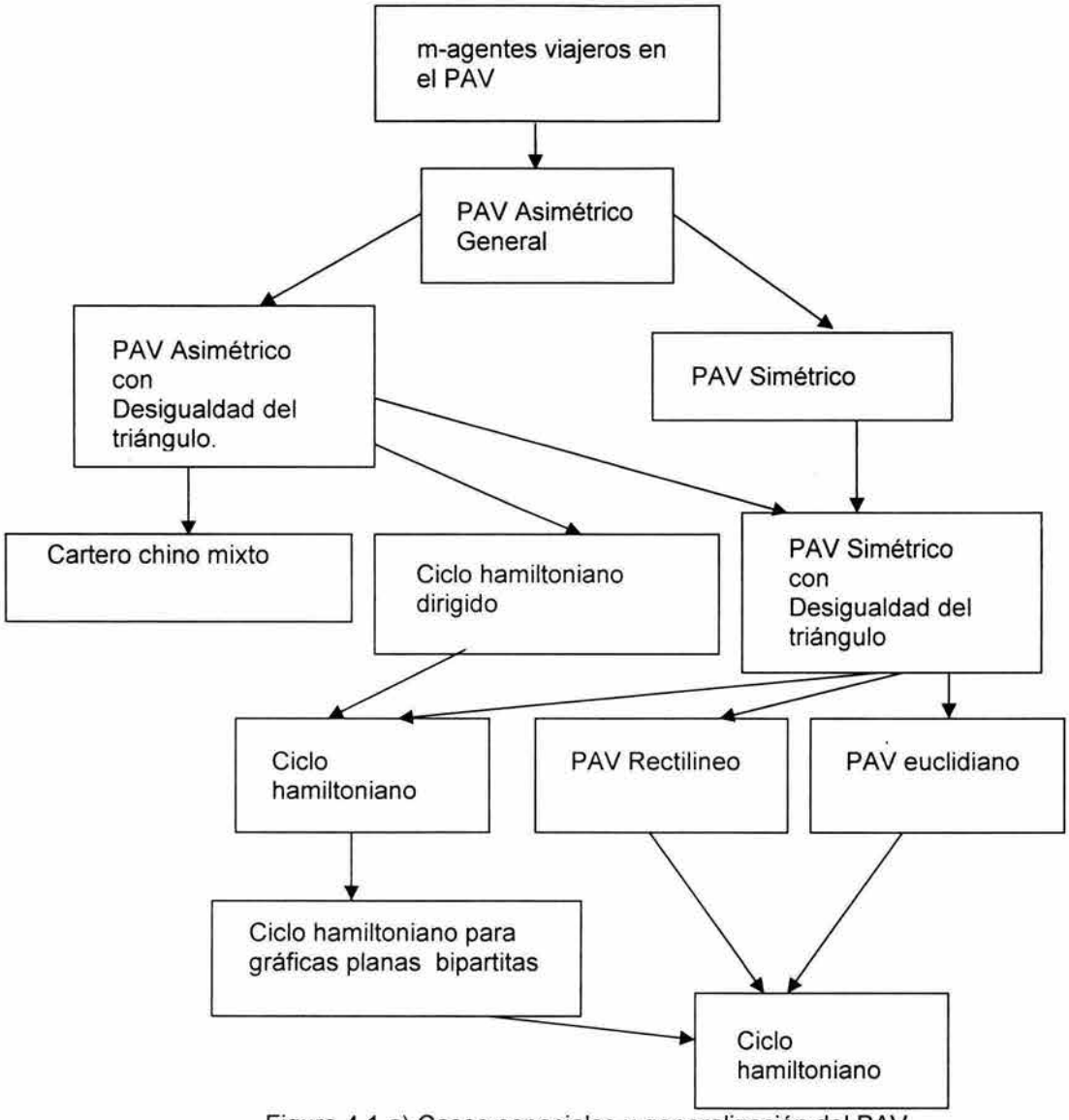


Figura 4.1 a) Casos especiales y generalización del PAV

4.1.3 Determinar el número de nodos

Este número se percibe en función de la cantidad de ciudades que quieres que visite el agente viajero. Este es un punto crucial porque, como se comentaba anteriormente, si es mayor de 10, debes pensar en un software y el algoritmo apropiado. En este caso se pueden usar algoritmos de solución exacta y más aún algoritmos heurísticos.

4.1.4 Software a utilizar

El software que se ha diseñado para resolver el problema del agente viajero está en función de la estructura con la que cuenta y principalmente se pueden dividir en problemas de agentes viajeros simétricos y asimétricos. En realidad el problema del agente viajero simétrico es una variante del problema del agente viajero asimétrico.

Si estamos interesados en únicamente saber cómo funcionan los algoritmos, en especial los algoritmos heurísticos, existen en la red demos que nos ayudan a entender cómo funcionan paso a paso estos algoritmos.

O bien tienes la opción de realizar tus propios algoritmos y programarlos en un lenguaje en especial, por ejemplo C ó C++, por mencionar algunos e incluso puedes hasta diseñar tu propia interfaz con JAVA, o en cualquier otro programa. En fin esto lo puedes realizar siempre y cuando tengas nociones de programación y el conocimiento de los algoritmos de solución.

A continuación presento ligas de interés en donde se puede encontrar software y que esta disponible para el público en general. Las siguientes ligas fueron tomadas tal cual de [Gutin, 2002]

Algoritmos exactos para el problema del agente viajero

En esta sección se proponen algoritmos que encuentran la solución óptima.

Concorde: Este es un software escrito en código ANSI C para algoritmos de solución exactos para resolver el PAV simétrico o el STSP y está disponible de manera gratuita para el público en general. El código implementa el algoritmo de ramificación y acotamiento (Branch and bound) desarrollado por Applegate, Bixby, Chvátal y Cook. También presenta bondades extras como la opción de resolver el problema con técnicas heurísticas, algoritmos para resolver problemas de redes; y también resuelve el problema de múltiples agentes viajeros. Este software se localiza en: tar.gz (``co991215.tgz'') del sitio web:
http://www.math.princeton.edu/tsp/concorde.html

CDT: Este software esta escrito en FORTRAN 77 del que se obtienen soluciones exactas del problema del agente viajero asimétrico. El código implementa el algoritmo de ramificación y acotamiento con algunas modificaciones en el algoritmo realizados por Carpaneto, Dell'Amico y Coth. El código se obtiene en formato gz (``750.gz'') del sitio web: http://www.acm.org/calgo/contents/

TSP1: Esta escrito en Turbo Pascal para resolver el PAV Simétrico y fue desarrollado por Volgenant y Van Den Hout. El código implementa el algoritmo de ramificación y acotamiento con algunas variaciones. También utiliza una implementación del algoritmo heurístico de Christofides mejorado con la técnica 3-Opt. El código se ubica en formato zip ("volgenan.zip") del sitio web: http://www.mathematik.uni-kl.de/~wwwwi/WWWWI/ORSEP/contents.html

tsp_solve: Esta escrito en C++ y resuelve de manera exacta el problema del agente viajero simétrico y asimétrico. Utiliza el algoritmo de ramificación y acotamiento basado en 1-tree, relajaciones de arborescencia y algoritmos heurísticos. Desarrollado por Hurwitz y Craig. Se puede obtener de manera gratuita en el siguiente sitio web: http://www.cs.sunysb.edu/~algorith/implement/tsp/implement.shtml

SYMPHONY: Este software fue realizado para resolver problemas de redes principalmente, esta escrito en ANSI C por Ralphs. Desarrollado principalmente para resolver problemas de ruteo. Se puede obtener de forma gratuita en: http://www.branchandcut.org/

tsp*: Es un código AMPL y resuelve problemas de agentes viajeros simétricos. Los códigos forman parte de un sitio web que contiene herramientas computacionales para resolver problemas de optimización combinatoria. Que contiene una implementación en AMPL del algoritmo heurístico Christofides. El paquete incluye códigos para encontrar árboles de expansión mínimos euclidianos. También incluye algunas herramientas para ver soluciones en problemas euclidianos (2-D) con Matemática. El programa se puede obtener en el siguiente sitio web: http://www.ms.uky.edu/~jlee/jlsup/jlsup.html

Combinatorica: Este programa es un agregado del paquete Matemática desarrollado por Skiena. Y resuelve el problema del agente viajero simétrico y el problema de encontrar circuitos hamiltonianos. Este programa se puede obtener en el siguiente sitio: ftp://ftp.cs.sunysb.edu/pub/Combinatorica/

rank: Este es un software que nos brinda intervalos de solución para los problemas de asignación . Así que este programa se puede usar para resolver problemas de agentes viajeros asimétricos. Es un archivo ejecutable escrito en DOS desarrollado por Metrick y Maybee. Y se puede obtener en:
http://www.mathematik.uni-kl.de/~wwwwi/WWWWI/ORSEP/contents.html

babtsp: Es un código realizado en Pascal y utiliza el método de ramificación y acotamiento desarrollado por Little, Murty, Sweeney, y Karen. Y desarrollado por Syslo para el libro realizado por este autor y DEo y Kowalik. El código se puede obtener en el siguiente sitio web:
http://www.mathematik.uni-kl.de/~wwwwi/WWWWI/ORSEP/contents.html

Algoritmos de aproximación para el problema del agente viajero

En esta sección se consideran una variedad de software que utilizan algoritmos heurísticos para resolver problemas de agentes viajeros simétricos y asimétricos.

LKH: Es un código escrito en código ANSI C y nos ayuda a resolver agentes viajero simétricos. Utiliza la técnica Lin-Kernighan desarrollado por Helsgaun. El código se puede obtener en un formato tar.gz (``LKH-1.0.tgz") y besta disponible de manera libre en: http://www.dat.ruc.dk/~keld/

SaCEC: Este es un software que nos ayuda a resolver el problema del agente viajero simétrico. Utiliza el algoritmo "Stem-and-Cyle Ejecution Chain" desarrollado por Rego, Glover y Gamboa. El software esta disponible en Microsoft Windows, y en plataformas de Linux y Unix en el sitio web:
http://faculty.bus.olemiss.edu/crego/

Concorde: Como se mencionó en la sección anterior este paquete incluye implementaciones muy efectivas en ANSI C para los siguientes algoritmos de aproximación. A) Heurística de Lin-Kernighan, b) heurística k-opt con $k= 2,2.5,3$ y c) Greedy, Nearest Neighborg, Boruvka, inserción mas lejana.

DynOpt: Es una implementación de un algoritmo basado en programación dinámica desarrollado por Balas y Simonetti. Y nos permite resolver problemas simétricos y asimétricos. El código se puede obtener en formato ASCII del sitio web:
http://www.andrew.cmu.edu/~neils/tsp/

LK: Es un código escrito en ANSI C. Utiliza el algoritmo heuristico Lin-Kernighan. El código se desarrolla usando "Cweb" y para leerlo requiere "LaTex" y "Cweb". El código utiliza la librería TSPLIB. El código se puede obtener en formato tar.gz del sitio web:
http://www.cs.utoronto.ca/~neto/research/lk

TSP: Es un código realizado en FORTRAN, y se utiliza el algoritmo de Christofides. Y se puede obtener por medio del libro [Lau, 1986]

Routing: Es un código realizado en FORTRAN y resuelve PAV asimétricos. Y se utiliza la técnica heurística 3-Opt . El código no esta disponible en la web, pero se pueden obtener en formato pdf en la siguiente dirección electrónica:
http://portal.acm.org/

twoopt, threeopt, fitsp: Es un código realizado en Pascal y resuelve PAV simétricos y está publicado en el libro de [Syslo, Deo y Kowalik, 1983]. Se utiliza el algoritmo 2-opt, 3-opt, y la heurística de inserción más rápida. El código se puede obtener el formato ("syslo.zip") en el siguiente sitio web:
http://www.mathematik.uni-kl.de/~wwwwi/WWWWI/ORSEP/contents.html

GATSS: Es un código realizado en GNU++ y nos ayuda a resolver PAV simétricos. Y utiliza una interface en HTML via CGIscript. El código implementa un algoritmo genético Standard. El código se puede obtener en un formato ASCII en el siguiente sitio web:
http://www.acc.umu.se/~top/travel_information.html

Tours: Este es un programa que nos ayuda a resolver PAV asimétricos y simétricos. El software es capaz de reportarnos límites superiores e inferiores. Y se puede utilizar el algoritmo de ramificación y acotamiento para instancias pequeñas, además tiene dispositivos gráficos y tiene la opción de salvar archivos en formato ASCII. Se incluye un manual de usuario para ayuda en línea. El software esta comercialmente disponible para plataformas en Microsoft Windows y se puede comprar y obtener información extra en:
http://www.isye.gatech.edu/people/faculty/Marc_Goetschalckx/

RAI: Es un código realizado en ANSI C y nos ayuda a resolver PAV asimétricos y utiliza el algoritmo de inserción aleatoria desarrollado por Brest y Zerovnik. El código se puede obtener de manera directa en el siguiente sitio web: http://marcel.uni-mb.si/~janez/rai

ECTSP: Este software contiene 2 heurísticas de solución para resolver PAV simétricos. La primer heurística esta basada en un algoritmo genético y la segunda en un algoritmo evolutivo. El paquete contiene dos archivos ejecutables para Windows 9x, y también contiene interfaces gráficas. El software ha sido desarrollado por Ozdemir y Embrechts y se puede obtener enviando correo a las siguientes dos direcciones electrónicas: (ozdemm,embrem}@rpi.edu).

GlsTsp: Es un código realizado en C++ y nos ayuda a resolver PAV simétricos y utiliza el algoritmo de búsqueda local desarrollado por Voudouris y Tsang. La versión actual 2.0 incluye dos archivos ejecutables para Microsoft Windows y Unix/Linux. El código se puede obtener de manera directa en el siguiente sitio web: http://cswww.essex.ac.uk/CSP/glsdemo.html

TSPGA: un código realizado en ANSI C y nos ayuda a resolver PAV simétricos y utiliza el paquete "pgapack". El código implementa el algoritmo de Frick el cual combina algoritmos evolutivos y busqueda local. El código se puede obtener escribiendolo al autor en el siguiente correo elctrónico: afr@aifd.uni-karlsruhe.de y se puede encontrar información adicional en el siguiente sitio web: http://www.stud.uni-karlsruhe.de/~ul63
Operations_Research_2.0: Es un software basado en Matemática y nos ayuda a resolver problemas de investigación de operaciones. Y los algoritmos heurísticos que utiliza son recocido simulado y la metaheurística ant-colony. También contiene el algoritmo de ramificación y acotamiento. El software se puede comprar en el siguiente sitio web: http://www.softas.de/op_researchframe.htm

Aplicaciones en JAVA

Las aplicaciones en JAVA son muy útiles, ya que nos ayudan a visualizar cómo trabajan los algoritmos paso a paso de manera gráfica, lo cual puede resultar muy didáctico. Algunos de los sitios siguientes contienen aplicaciones en JAVA y utilizan algoritmos heurísticos mejor conocidos.

- http://mathsrv.ku-eichstaett.de/MGF/homes/grothmann/java/tsp.html
- http://home.planet.nl/~onno.waalewijn/tspx.html
- http://www.wiwi.uni-frankfurt.de/~stockhei/touropt.html
- http://itp.nat.uni-magdeburg.de/~mertens/TSP/index.html

4.1.4 Elegir el tipo de solución

Aquí tienes que elegir que tipo de solución quieres. Si la quieres exacta (óptima) o bien cercana al óptimo.
Si quieres una solución exacta debes recurrir a las técnicas exactas y si deseas una solución aproximada puedes acudir a las técnicas heurísticas. En realidad esta decisión depende de la rapidez con la que quieres la respuesta y el número de nodos con las que cuenta el problema.
En general para cualquier escenario que se plantee se recomienda el uso de las técnicas heurísticas, dado que te dan una solución muy rápida y muy cercana al óptimo, como lo demuestran diversos estudios. [Castañeda, 2000] y [Jünger, Reinelt y Rinaldi, 1995].

4.1.5 Obtención de la solución

Si elegiste la solución las técnicas heurísticas como método de solución seguramente se obtendrá la solución de manera inmediata, pero si, elegiste las técnicas exactas, se tendrá que esperar al menos unas horas si el software es muy eficiente, y claro en el mejor de los casos el software te da la respuesta de manera inmediata, y esto ocurre, cuando la técnica exacta se combina con alguna técnica heurística, como fue es el caso del software ARCVIEW3.2.

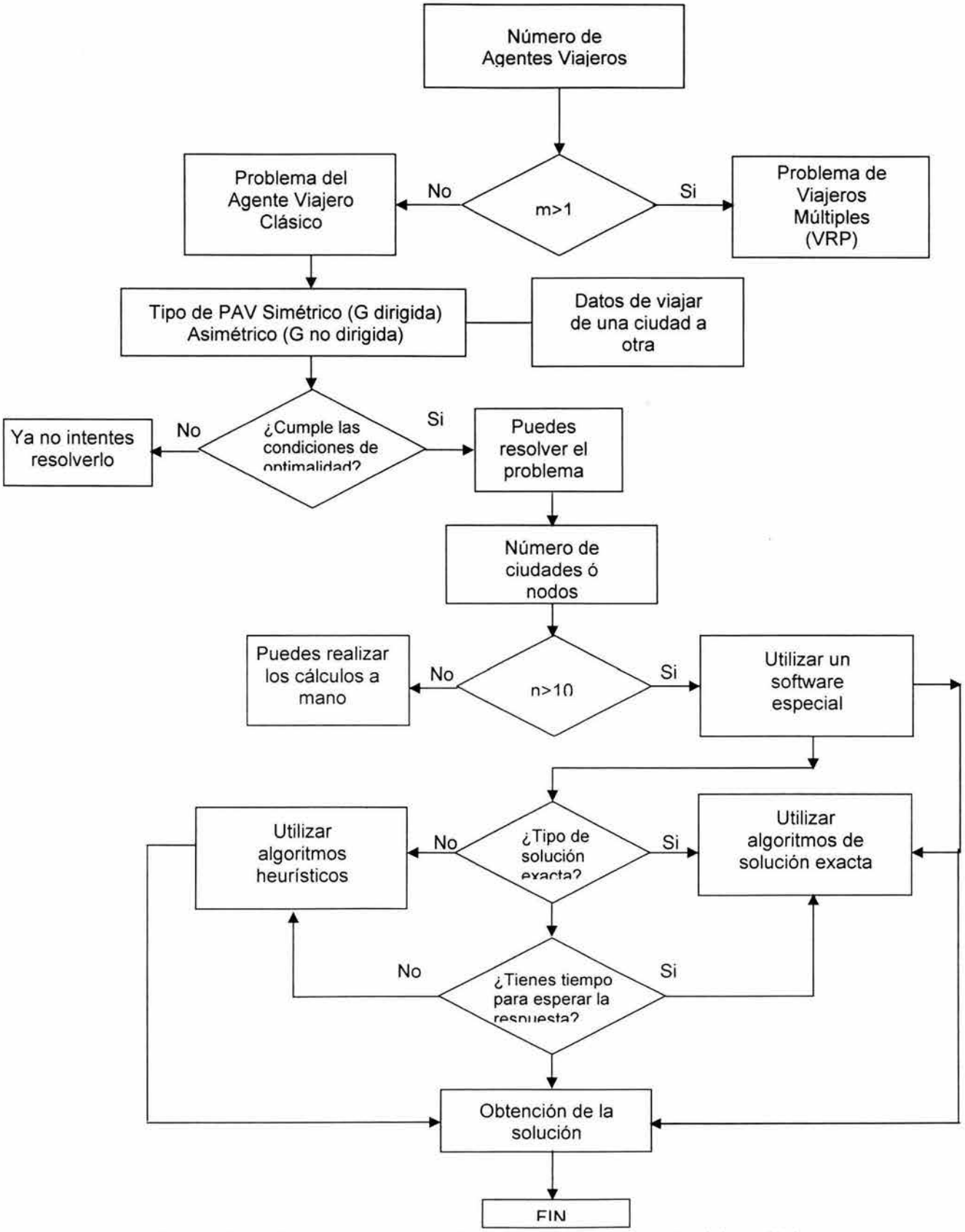


Figura 4.1. b) Metodología para resolver el problema del agente viajero clásico

Básicamente estos son los pasos que tienes que realizar para resolver un problema del agente viajero. En la siguiente sección se muestran dos estudios comparativos para que el lector se de una idea de la eficiencia con la que trabajan los algoritmos, tanto de aproximación como de los métodos exactos.

4.2 Estudios comparativos

En esta sección se muestran en especial dos estudios comparativos en donde se muestra la eficiencia de los algoritmos de aproximación y algoritmos exactos, en el caso de [Castañeda ,2000] y un estudio comparativo de algoritmos heurísticos [Jünger, Reinelt y Rinaldi, 1995].

4.2.1 Estudios comparativos de optimalidad de algoritmos exactos y heurísticos

La mayoría de los estudios comparativos se centran en dos cosas: en la calidad de la solución y el tiempo de solución. Y se han realizado estudios en donde se compara la eficiencia de los algoritmos exactos contra los algoritmos heurísticos, encontrando resultados verdaderamente sorprendentes. Un estudio lo realizó Castañeda Roldán Yolanda en su tesis para obtener el grado de maestro en ciencias computacionales en la Universidad De Las Américas, Puebla (UDLAP) en el año 2000. En este estudio se muestra la comparación de los siguientes algoritmos heurísticos: Dos Optimal (2-Opt), Adaptación Prim, Híbrido Dos Optimal-Prim y Red Neuronal de Hopfield contra el algoritmo exacto: Una mejor Ramificación y Acotamiento (A Better Branch and Bound).

Se realizaron tres pruebas o estudios en esta tesis: Comparación de exactitud de técnicas exactas contra un heurístico, el segundo un análisis estadístico para saber qué tan confiable es el resultado; y en el tercero se elaboró un análisis del tiempo de ejecución con respecto al número de ciudades.

4.2.1.1 Comparación de exactitud de técnicas exactas contra técnicas heurísticas [Castañeda, 2000]

Como sabemos, existen diferentes metodologías para saber qué tan eficiente es un algoritmo con respecto a otro. En este caso se comparó la eficiencia de los algoritmos con respecto de un algoritmo exacto; en este caso, (A Better Branch and Bound), usando la métrica teórica estándar para medir la calidad de un algoritmo de aproximación, se calculó del Factor de Garantía r_A y el Cociente de aproximación C_A (ver la página 6 del capítulo 3). Esto se debe a que es necesario analizar qué tanto se aproximan los algoritmos aproximados a su solución real.

El primer estudio se realizó con 3 y hasta 12 ciudades, y de este estudio se concluye lo siguiente:

En las figuras de 4.2 a 4.5 se observa que, para la muestra de 10 problemas PAV, tanto para el método Dos-Optimal como el Híbrido Dos Optimal-Prim, 8 problemas PAV dan resultados exactos y dos de ellos dan el resultado aproximado. En contraste con el método Adaptación de Prim y Redes Neuronales de Hopfield, que de 10 problemas PAV, sólo 3 problemas PAV dan resultados exactos y 7 de ellos dan resultados aproximados. Esto sugiere que los dos primeros tienden hacia una mejor aproximación que los dos últimos.

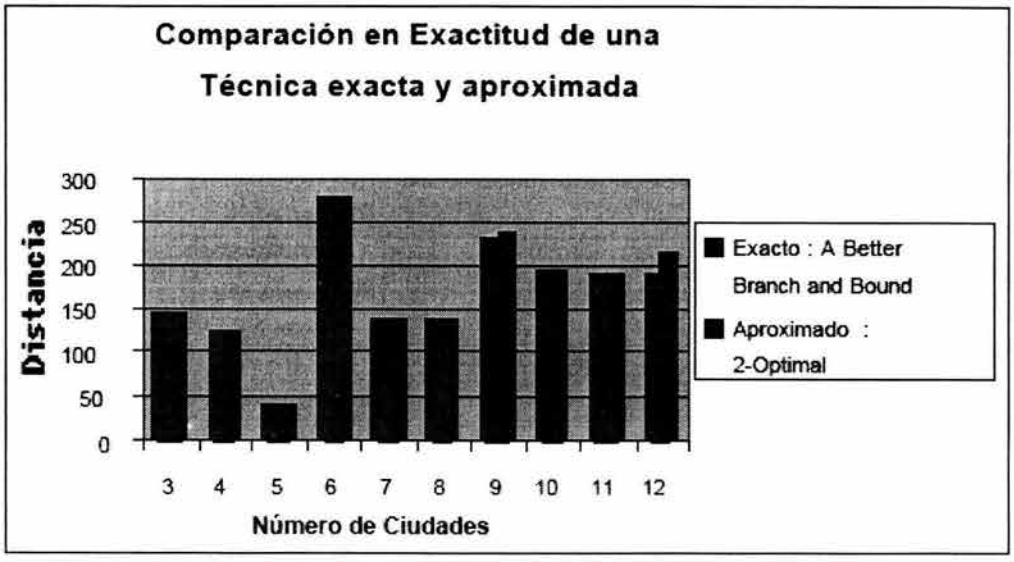


Figura 4.2 Comparación en exactitud de A Better Branch and Bound y 2-optimal

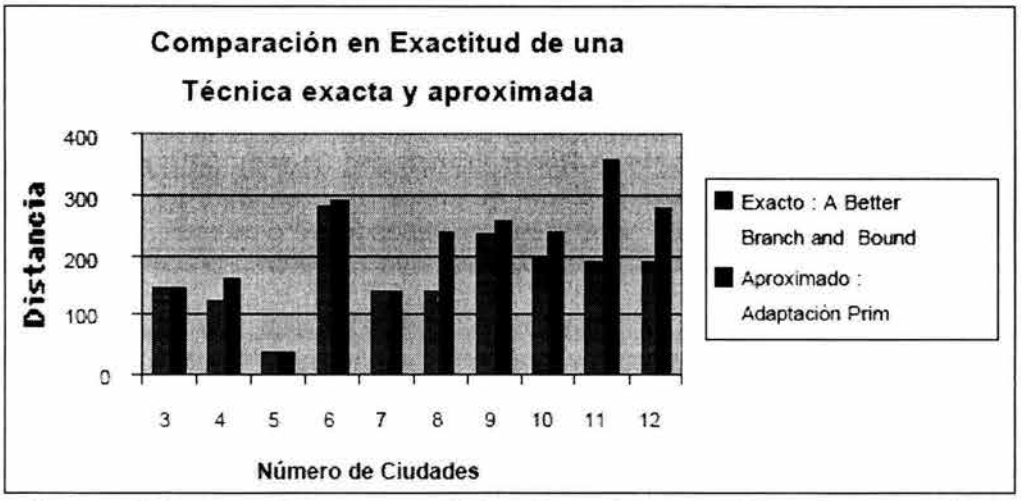


Figura 4.3 Comparación en exactitud de A Better Branch and Bound y adaptación Prim.

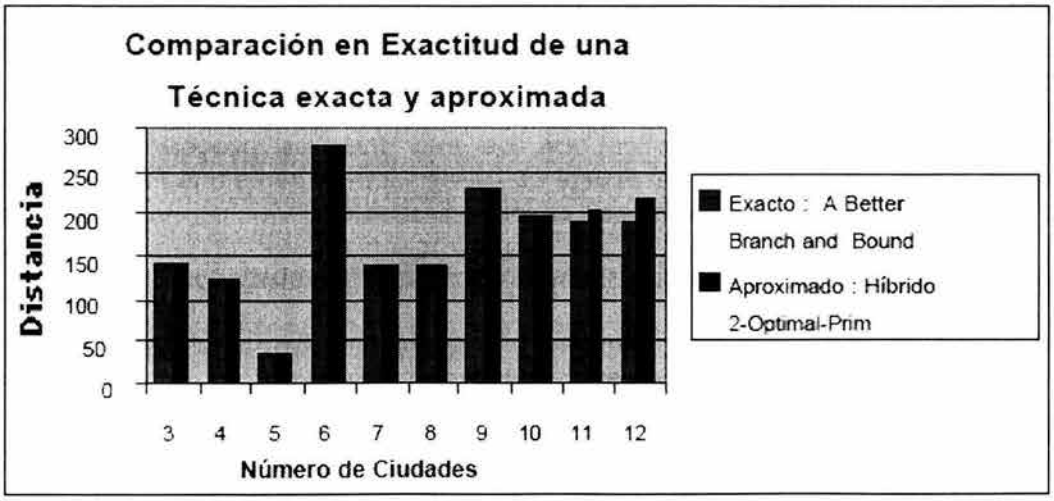


Figura 4.4 Comparación en exactitud de A Better Branch and Bound e híbrido 2-optimal-Prim

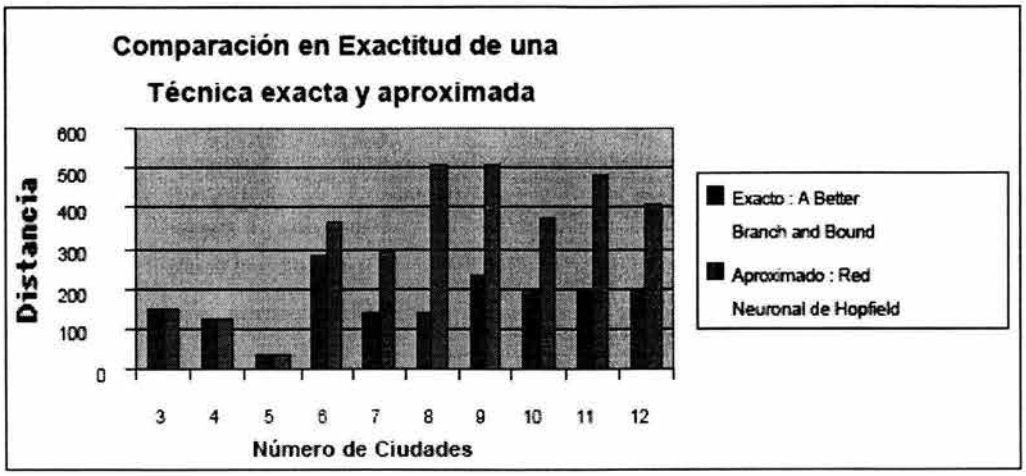


Figura 4.5 Comparación en exactitud de A Better Branco and Bound y Red Neuronal de Hopfield

Análisis estadístico

En este caso se realizaron pruebas de bondad de ajuste, ya que ésta nos proporciona una herramienta para probar si un grupo de datos se aproxima o semeja a una distribución conocida. En este caso se quería comprobar que los datos arrojados por los métodos aproximados tienen un comportamiento estadísticamente semejante al del método exacto Una Mejor Ramificación y Acotamiento (A Better Branch and Bound). Esto permite asegurar que estos métodos aproximados son bastante cercanos a la realidad. La confiabilidad de estos métodos dependerá del valor alfa; que corresponde al grado de confiabilidad. El número de observaciones por prueba fue de 18, porque se corrieron 3 problemas PAV de 10, 12,.., 20 ciudades. Y se obtuvo lo siguiente:
De la heurística Dos Optimal : No se puede rechazar la hipótesis, por lo que se dice que tienen una misma distribución con un grado de confianza mayor de 99.5 % y un riesgo menor de 0.5 %.
Adaptación Prim: No se puede rechazar la hipótesis, por lo que se dice que tienen una misma distribución con un grado de confianza entre 99.5 % y 97.5%y un riesgo entre 0.5 % y 2.5 %.
Híbrido 2-Optimal: No se puede rechazar la hipótesis, por lo que se dice que tienen una misma distribución con un grado de confianza mayor de 99.5 % y un riesgo menor de 0.5 %.
Red Neuronal de Hopfield: Se rechaza la hipótesis y se afirma que no tienen la misma distribución por tener un grado de confianza menor de 5%.

Computacionalmente hablando significa que los tres primeros pueden usarse con la confianza de que la exactitud de la respuesta es buena. Con respecto a la Red Neuronal de Hopfield, debe mejorarse la red para que se mejore en exactitud.

Análisis del Tiempo de Ejecución con respecto al número de ciudades

En este estudio se corrieron los algoritmos y se comparó el tiempo de solución de una técnica heurística contra una exacta (A better Branch and Bound)

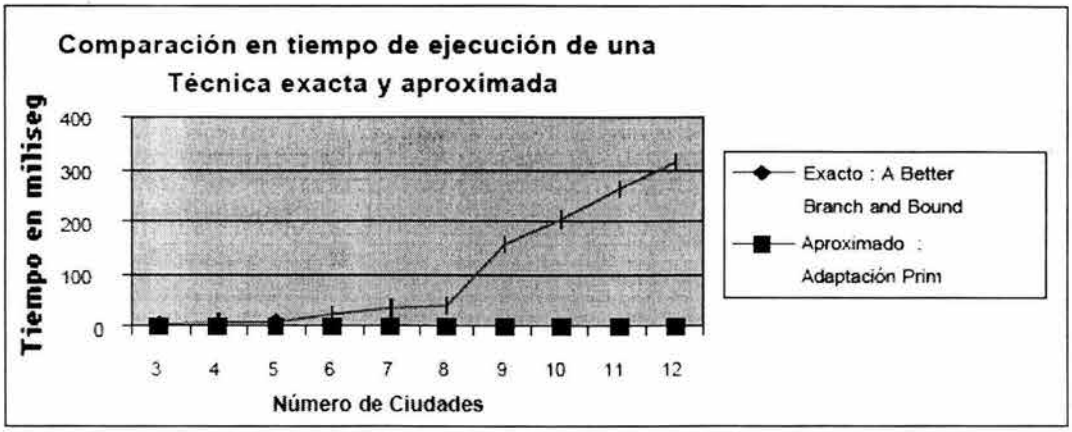


Figura 4.6 Comparación en tiempo de ejecución del algoritmo 2-opt y a Better Branch and Bound

Figura 4.7 Comparación en tiempo de ejecución del algoritmo Adaptación Prim y a Better Branch and Bound

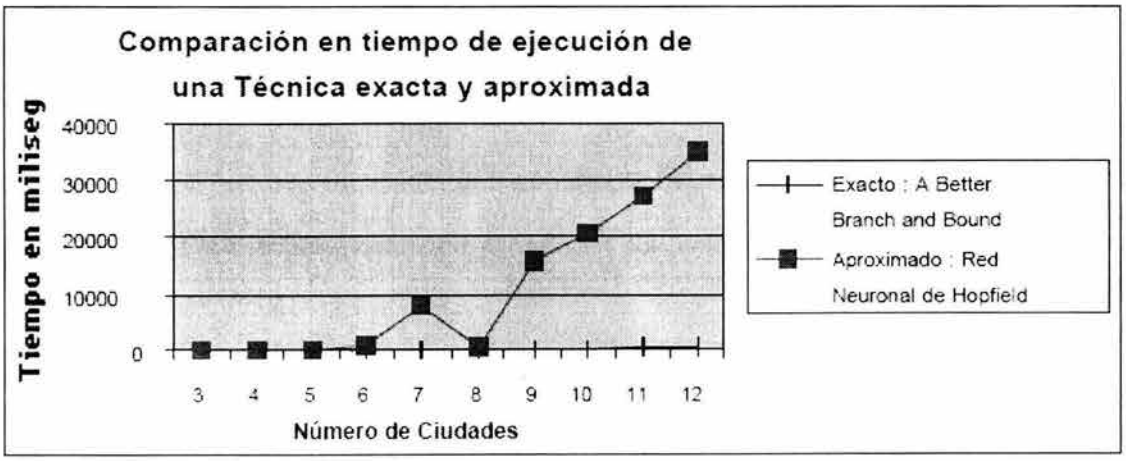


Figura 4.8 Comparación en tiempo de ejecución del algoritmo híbrido 2-optimal-prim y a Better Branch and Bound

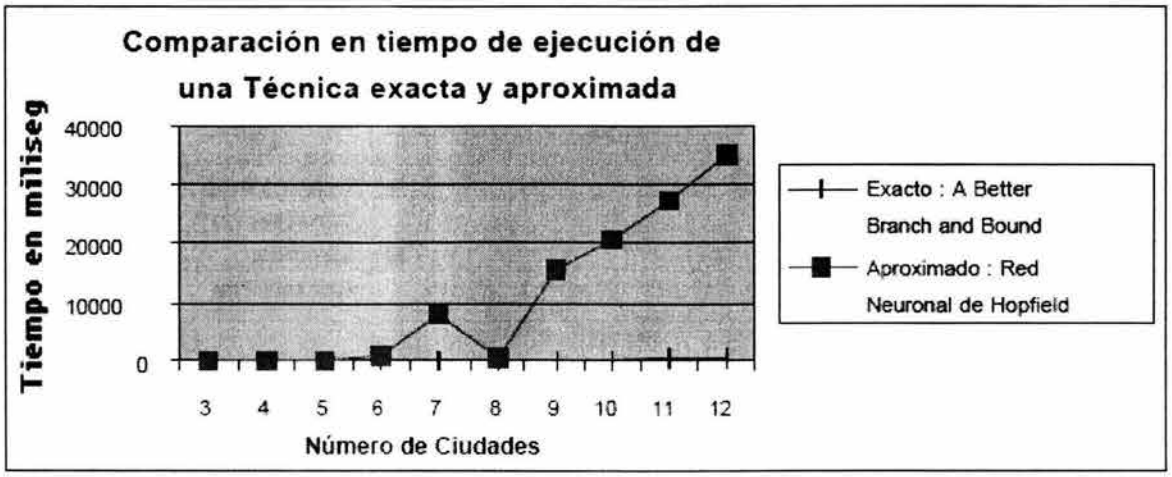


Figura 4.9 Comparación en tiempo de ejecución del algoritmo red neuronal de hopfield y a Better Branch and Bound

De la figura 4.6 a 4.9 sugieren la utilización de diferentes tipos de técnicas de solución dependiendo del número de ciudades. Para pocas ciudades (hasta antes de 8) es posible (ya que se trata de miliseg) utilizar métodos exactos dado que el tiempo de respuesta es relativamente pequeño y tiene como ventaja sobre un aproximado que la solución es la exacta. Y con un crecimiento mayor de 10 en el número de las ciudades la diferencia en tiempo aumenta, por lo que son mejores los métodos aproximados. Cabe señalar que siempre la curva de los aproximados (a excepción de la Red Neuronal de Hopfield) se prolonga por abajo de la curva de un método exacto, significando que el tiempo de respuesta será menor que la respuesta de una técnica exacta, aunque en algún momento la respuesta será aproximada. La Red Neuronal de Hopfield requiere de tiempos mayores del resto de los aproximados; pero, en este punto, cabe aclarar que la red Neuronal de Hopfield se corrió con los parámetros default de Hopfield por facilidad en el muestreo. Pueden mejorarse los resultados en exactitud y tiempo si se juega con los parámetros, es decir, se pueden variar los parámetros iniciales de la red a otros diferentes de los default de Hopfield.

Además, debe tomarse en cuenta que la bondad del método de la Red de Hopfield sobre el resto de los métodos aproximados es que éstos usan siempre una ruta inicial, la cual mejora en gran medida el algoritmo. La Red Neuronal de Hopfield, por el contrario, parte de cero, es decir, sin ruta alguna; sin embargo, encuentra siempre una ruta válida.

En este sentido cabe mencionar que los algoritmos heurísticos que se han desarrollado para el agente viajero son bastante confiables y además muy rápidos y que los algoritmos al hibridarse con otro método de solución, los resultados son sorprendentes al tener un margen de optimización con respecto al óptimo de 1% a 3%.

4.2.1.2 Estudio comparativo de técnicas heurísticas [Jünger, Reinelt y Rinaldi, 1995]

Otro estudio realizado por [Jünger, Reinelt y Rinaldi, 1995]. Se comparan los siguientes algoritmos heurísticos: Vecino más cercano, inserción más lejana, christofides y ahorros de los algoritmos de construcción, también se evaluaron algoritmos de mejora como:2-opt., 3-opt.,Lin y Kernighan. Se compararon 30 agentes viajeros tomados de TSPLIB y se muestran únicamente los resultados del problema más grande de 2392 ciudades. Y se conoce como pr2392.

El método de análisis de la eficiencia de los algoritmos fue comparándolos con una solución exacta, en este caso se utilizó el algoritmo rama y corte.
En su estudio compararon técnicas heurísticas de construcción y de mejora.

En esta primera tabla se muestra la desviación del óptimo de los algoritmos de construcción o constructivos del problema del agente viajero que más se utilizan en la actualidad, como son: el vecino más próximo, el de inserción más lejana, el algoritmo de Christofides y el algoritmo de ahorros.

Heurístico	Desviación del óptimo	Tiempo de Ejecución
Vecino más próximo	18.6%	0.3
Inserción más Lejana	9.9%	35.4
Christofides	19.5%	0.7
Ahorros	9.6%	5.07

Tabla 4.1 Comparación de desviación del óptimo de algoritmos heurísticos de construcción

Se puede observar que los tiempos de computación son muy parecidos entre sí e inferiores a 1 segundo en promedio.
A la vista de los resultados podemos concluir que, tanto el método de los ahorros como el de inserción basado en el elemento más lejano son los que mejores resultados obtenidos, aunque presentan un tiempo de computación mayor que los otros dos.

Heurístico	Desviación del óptimo	Tiempo de Ejecución
2-óptimo	8.3%	0.25
3-óptimo	3.8%	85.1
Lin y Kernighan 1	1.9%	27.7
Lin y Kernighan 2	1.5%	74.3

Tabla 4.2 Comparación de desviación del óptimo de algoritmos heurísticos

Respecto de los tiempos de ejecución, se observa como al pasar de una exploración 2-opt a 3-opt aumenta considerablemente el tiempo de ejecución. Y también que el algoritmo Lin y Kernighan tiene una desviación del óptimo menor al algoritmo 2 y 3 opt.
A la vista de los resultados, parece más interesante utilizar movimientos compuestos, que permitan controlar movimientos simples de no mejora, que utilizar movimientos k-óptimos con valores altos de k que, por su complejidad, consumen mucho tiempo y, sin embargo, no llegan a tan buenos resultados.

4.3 Aplicaciones

Aplicaciones en la investigación de operaciones se encuentran prácticamente en todos los niveles y en todo tipo de industrias. Es evidente que las corporaciones aspiran a tomar decisiones que les reditúen beneficios económicos, y normalmente, estas decisiones se encuentran restringidas de forma muy compleja. Estos atributos son únicos de modelos de investigación de operaciones. En las últimas décadas el impacto de la investigación de operaciones en la industria ha sido impresionante, convirtiéndose en ganancias (o ahorros) con frecuencia multimillonaria en los diversos ramos industriales.
El problema del agente viajero es la base para muchas aplicaciones, en especial, en la de reparto de mercancía en una ciudad. Las situaciones reales suelen complicarse con más de un vehículo de reparto; horarios de reparto impuestos por los clientes; demoras en los lugares de entrega; capacidades en los vehículos, que limitan los recorridos, etc.
En las siguientes líneas se hablará de una de las aplicaciones más comunes del problema del agente viajero que es la programación de tareas en una máquina.

4.3.1 Programación de tareas en una máquina

Algunas veces, en algún taller de manufactura, se cuenta con una sola máquina en la cual se procesan diferentes tareas, una a la vez. Ahora bien, para procesar cada una de estas tareas la máquina requiere de cierta configuración característica de la tarea, pueden ser: número y tamaño de diferentes dados, colocación de cuchillas a cierta distancia unas de otras, colorantes para alguna fibra, etc. De manera que una vez que una tarea ha sido terminada, es necesario preparar la máquina para procesar una nueva tarea, aquí será necesario invertir un cierto tiempo, y este tiempo dependerá de la tarea recién procesada y de la próxima. Si las características de una tarea son similares a las de otra, se piensa que el tiempo que se requiere para pasar de una configuración a otra será pequeño, en comparación del tiempo requerido para ir de una tarea a otra con características muy diferentes.
Desgraciadamente durante las labores de preparación de la máquina, ninguna de las tareas se puede ejecutar, así que este tiempo es tiempo perdido, y se está desaprovechando la capacidad de la máquina, esto representa un costo de oportunidad para la empresa. Es importante entonces encontrar el orden en el cual se deben de procesar estas tareas, con el fin de reducir al mínimo el tiempo perdido.
Aún cuando este problema parezca tener ninguna relación con el PAV, se puede formular de la misma manera. Cada tarea puede ser vista como una de las ciudades a visitar y el tiempo necesario para cambiar la configuración de la máquina corresponde a la distancia que hay entre una ciudad y otra. Encontrar la manera de ordenar las tareas para minimizar el tiempo total de preparación es equivalente a diseñar la ruta, esto es, el orden en el cual se deben visitar las ciudades para minimizar la distancia total recorrida. Esto nos da una idea de lo crucial que resulta tener buenas soluciones para el PAV en un ambiente de manufactura.

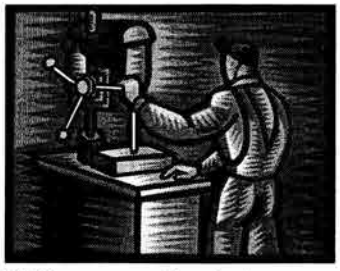


Figura 4.10 Programación de tareas en una máquina

4.3.2 Logística de distribución de mercancía a los clientes o VRP

Generalmente, algunas empresas que distribuyen bienes perecederos necesitan hacerlo en un tiempo corto, un esquema muy común es que la empresa disponga de un almacén central, en el cual se concentran los bienes a distribuir y una flotilla de unidades de transporte se encarga de visitar a los clientes para hacer entrega de la mercancía.
Analicemos los componentes de este problema, en primer lugar tenemos que las unidades de servicio son limitadas, la forma en la que se podría efectuar la entrega de mercancías en el menor tiempo posible, sería enviar una unidad a cada uno de los clientes. Pero lo más realista sería pensar que no se tienen tantas unidades como clientes. Ya que esto resultaría sumamente costoso. Si la empresa dispone de una sola unidad el costo fijo se reduce bastante, y el problema de determinar la ruta que debe de seguir el vehículo para entregar en el menor tiempo toda la mercancía es ni más ni menos el PAV. Pero aquí hay dos problemas en los que tenemos que pensar: en primer lugar, puede ser que el tiempo mínimo (si es que se puede determinar) resulte demasiado largo, por ejemplo si se trata de la entrega de leche, esta debe entregarse por la mañana, que es cuando los clientes la requieren, y con una sola unidad de entrega, podría darse el caso que los últimos clientes la fueran recibiendo por la tarde. Por otro lado, las unidades tienen una cierta capacidad de almacenamiento, y tal vez que se necesiten varias para cargar con la mercancía que debe ser entregada.
Así pues, vemos que este problema contiene dentro de sí muchos más. Primero: determinar cuál es el tamaño ideal de la flota de vehículos. Segundo: determinar cuáles son los clientes que deben ser asignados a cada unidad para hacer la entrega. Y finalmente, cuál es la ruta que debe seguir cada una con el fin de terminar con el reparto en el menor tiempo posible (PAV). Para complicar más las cosas estos problemas no son independientes, sino que la solución de uno determina la de otro. Este problema se conoce como problema de ruteo de vehículos (VRP: Vehicle Routing Problem). Figura 4.11

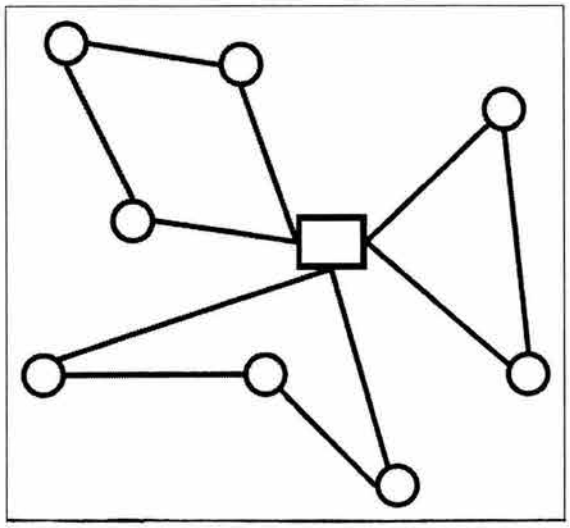


Figura 4.11 Ejemplo de un ruteo factible en un VRP (una central de abasto, ocho clientes y tres unidades de servicio)

4.3.3 Clasificación de secuencias de la molécula de ADN

El Problema de Clasificación de secuencias de la molécula de ADN es de suma importancia en la biología molecular. En éste se tiene una colección de fragmentos (subcadenas de alguno de los dos hilos de varias moléculas idénticas de ADN que se han "roto" en diferentes formas) y se pretende **deducir la secuencia de la molécula de ADN entera**. Este problema se modela de tres formas: supercadena común más corta (SCC ó SCS – Shortest Common Superstring), reconstrucción y multicontig (multiple contiguously covered regions ó múltiples regiones cubiertas contiguamente). En el modelo de la supercadena común más corta, el problema es **encontrar la cadena más corta que contenga cada fragmento como subcadena**. Si creamos un grafo donde cada vértice denote un fragmento y el peso de cada arista sea el traslape entre fragmentos, entonces el problema se resuelve encontrando un ciclo Hamiltoniano.

En el Departamento de Química y Biología de la Universidad de las Américas-Puebla se trabajó en la clasificación de secuencias de ADN. Se pretende tipificar una familia de virus que causan el cáncer cérvico uterino (los papilomavirus humanos (PVH)).

Los virus poseen una estructura de ADN cambiante (suelen mutar), por lo que se agrupan en familias. Si dos ADNs muestran 10% ó mayor diferencia, entonces pertenecen a tipos distintos. Si la variación es menor al 10%, pero mayor a 5%, entonces pertenecen al mismo tipo, pero a subtipos diferentes. Si la diferencia no alcanza el 5%, hablamos de variantes. En rara ocasión encontraremos una coincidencia exacta.

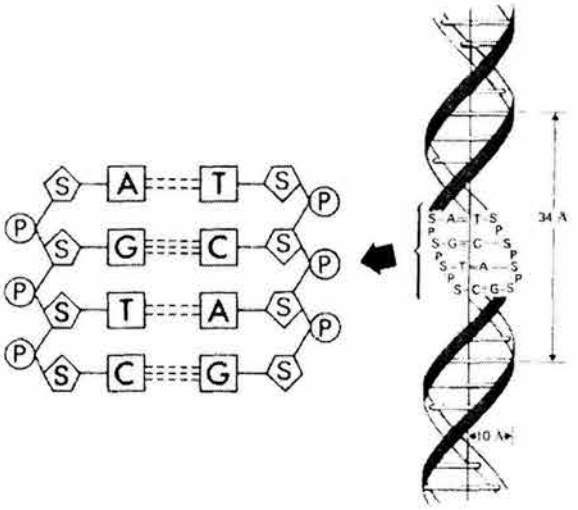


Figura 4.12 Molécula de ADN

La selección de la combinación óptima de costo mínimo sin que se repitan los patrones, es un problema semejante al Problema del Agente Viajero (PAV). Conociendo la distancia entre cualquier par de patrones se generan un conjunto de clases que los clasifiquen por similitud, y teniendo un número arbitrario de enzimas que generan diferentes clasificaciones, con cierto grado de información redundante entre cualquier par de enzimas; encontrar cuántas y cuáles enzimas, al combinar la información que me proporcionan, generan: una clasificación determinada ó la clasificación más fina posible. Tipificando una familia de virus que causan el virus del papiloma humano.

4.4 Ejemplo práctico

Con este ejemplo se pretende mostrar la metodología mostrada en un principio y que el lector pueda resolver un ejemplo sencillo: encontraremos un tour óptimo a través de las 31 ciudades principales de los Estados de la República Maxicana y el Distrito Federal. El agente viajero será cada uno de nosotros y dado que quiere viajar por toda la república optimizando su tiempo y su dinero es necesario encontrar el tour óptimo. Pero además queremos saber la respuesta lo antes posible, dado que se nos acaban las vacaciones y nuestro dinero!

México se encuentra situado en el norte del continente Americano, junto con Canadá y Estados Unidos de América; se localiza en el hemisferio occidental hacia el oeste del meridiano de Greenwich.

La extensión territorial del país es de 1'964,375 km², con una superficie continental de 1'959,248 km² y una insular de 5,127 km². Esta extensión lo ubica en el decimocuarto lugar entre los países del mundo con mayor territorio.

México colinda en su parte norte con los Estados Unidos de América, a lo largo de una frontera de 3,152 km y al sureste con Guatemala y Belice, con una frontera conjunta de 1,149 km de extensión. La longitud de sus costas continentales es de 11,122 km, por lo cual ocupa el segundo lugar en América, después de Canadá.

Hasta 1996 la red de carreteras tenía una longitud de 312,301 km. Si fuera posible colocar una carretera después de otra, se podrían dar más de siete vueltas a la Tierra sobre el Ecuador.
En la tabla se muestran cada uno de los 32 estados de la República Mexicana a través de los cuales deseamos pasar.

1. Mexicali	9. Cd. Victoria	17. Colima	25. Puebla
2. La paz	10. Zacatecas	18. Morelia	26. Tuxtla Guitérrez
3. Hermosillo	11. San Luis Potosí	19. Toluca	27. Villahermosa
4. Chihuahua	12. Tepic	20. Pachuca	28. Chilpancingo
5. Saltillo	13. Aguascalientes	21. D.F.	29. Oaxaca
6. Culiacán	14. Guanajuato	22. Tlaxcala	30. Campeche
7. Durango	15. Guadalajara	23. Xalapa	31. Mérida
8. Monterrey	16. Querétaro	24. Cuernavaca	32. Chetumal

Tabla 4.3 Estados de la República Mexicana

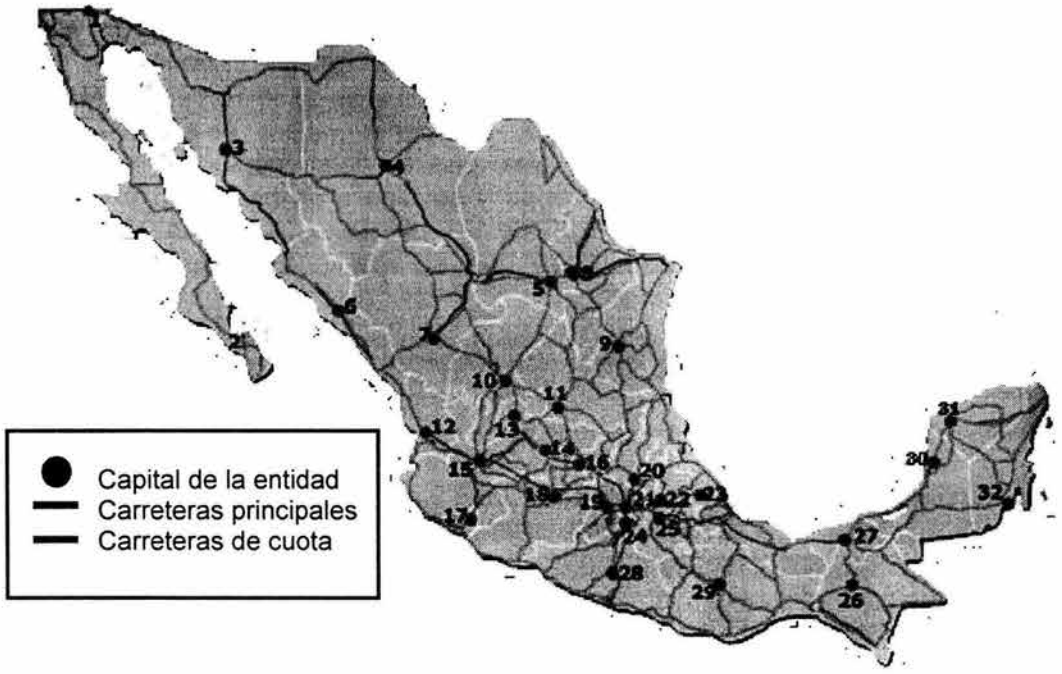


Figura 4.13 Carreteras principales de la República Mexicana

Siguiendo la metodología que se planteó en el principio primero tenemos que:

1. Determinar el **número de agentes viajeros** en este caso contamos con solo un agente viajero, que somos en este caso nosotros.

2. Después tenemos que determinar el **tipo de agente viajero**. El tipo de problema está en función de las características especiales que tenga nuestro agente viajero.

Podemos decir que es **un solo agente viajero** que es **simétrico** dado que ir de un estado por ejemplo de Zacatecas a Guadalajara es la misma distancia de viajar de Guadalajara a Zacatecas, considerando que utilizaríamos el mismo camino. Los **datos** son **enteros positivos** y representan las distancias en carretera de una ciudad a otra, por lo que tenemos la opción de insertar los datos en una matriz de adyacencia. Cabe señalar que las **únicas restricciones** son las asociadas al problema: no visitar una ciudad más de una vez y regresar a la ciudad de la cual partí. No hay restricciones adicionales. Porque podríamos restringirlo, tomando una ciudad como punto de partida y además sujetar al viajero a visitar las ciudades en un cierto orden, pero este no es el caso.
O bien tenemos la opción de modelarlo como un problema de programación matemática en especial como un problema de programación entera binario. Como se muestra a continuación. Y de manera paralela necesitamos saber como se obtuvieron nuestros datos. En este caso los datos se obtuvieron del Guía Roji, en donde se muestran las distancias en carreteras de un estado a otro. Se pueden observar los datos a partir de una matriz de adyacencia, ésta se puede observar al final de este capítulo.

$$\min Z = \sum_{i \in V} \sum_{j>i} c_{ij} x_{ij}$$

s.a.

$$\sum_{j<i} x_{ij} + \sum_{j<i} x_{ij} = 2 \qquad i \in V$$

$$\sum_{i \in S} \sum_{j \in S, j>i} x_{ij} \leq |S| - 1 \qquad \forall S \subset V, S \neq 0$$

$$x_{ij} = 0,1 \qquad \forall i, j \in V$$

Y dado que el problema cumple con las condiciones de optimalidad: para que tenga solución el PAV simétrico una solución es necesario que, la cardinalidad de cada nodo sea mayor o igual a dos. En el caso de gráficas dirigidas, la condición necesaria es que el grado de entrada y el de salida de cada nodo deben ser mayor o igual que uno. Y sabemos que tiene solución óptima dado que la matriz cumple la desigualdad del triángulo, es decir, que no existe ningún arco (digamos *i* y *j*) cuyo costo es infinito; pues la suma de los costos de la trayectoria que une *i* y *j* tiene que ser mayor que el costo asociado al arco (*i, j*) Por tanto, la gráfica es completa y existe solución factible (Por ejemplo, $\{(1,2),(2,3),...,(n,1)\}$. Como el número de soluciones factibles es finito, implica que existe una solución óptima.

3. Determinar el **número de nodos**, pues bien, el número de nodos es de 32.

4. Software a utilizar: Siguiendo el paso anterior dado que queremos una solución con respuesta rápida y además exacta, se hace necesario el uso de un software. En realidad existe software que nos puede ayudar a resolver estos problemas como lo es LINDO o WINQSB, o el programa ARCVIEW en su versión 3.2, además de los listados en las secciones anteriores. Son programas que son muy accesibles y soportan un gran número de nodos.

El programa WINQSB consta de 19 módulos entre los cuales figura el módulo de programación de redes o Network Modeling NET, que es el módulo que resuelve problemas de redes que nos ayuda a resolver el problema de transbordo , transporte, asignación, camino más corto, flujo máximo, árbol de expansión mínima y el problema del agente viajero.

Las capacidades específicas de NET incluyen:

Algoritmo para resolver problemas de rutas más cortas, árboles de expansión mínimos, ramificación y acotamiento y las heurísticas: el vecino más cercano, incersión más barato y 2-opt. Además despliega paso a paso el método Hungaro, mediante soluciones gráficas, análisis paramétrico, encuentra soluciones alternativas para los problemas de transporte. Tiene la facilidad de introducir datos en forma matricial o en forma gráfica.

Tiene la bondad de tolerar un gran número de nodos 2050 para ser más exactos. Cabe señalar que el módulo Network Modeling toma como punto de partida el nodo que se inserte primero, en específico para la heurística del vecino más cercano. Es decir, si el nodo 1 es la ciudad de Aguascalientes, entonces este algoritmo, tomará al estado de Aguascalientes como la ciudad de la cual se tiene que partir y a la cual regresar.

En el caso de LINDO es un programa bastante accesible, pero no nos permite la opción de ver el problema gráficamente y tampoco nos permite el uso de algoritmos heurísticos, es decir, resuelve el PAV de manera exacta a través de la programación matemática utilizando el método simplex como método de solución. Lo cual limita nuestras posibilidades de utilizar otro tipo de algoritmos como los heurísticos y sería necesario plantear el problema como un problema de programación entera binaria, ya que los planteamientos de los algoritmos exactos requieren una estructura de este tipo.
Otro software que se utilizó fue ARCVIEW versión 3.2. Este software es de los sistemas que se les llama GIS "Sistemas de Información Geográfica" y sirve para realizar distintos estudios que utilizan redes geográficas, por ejemplo, calles, autopistas, ríos, líneas eléctricas, etc. Uno de ellos es el cálculo de rutas más cortas. Este software cuenta con distintas extensiones que ayudan a resolver varios tipos de problemas, la extensión que nos ayuda a resolver los problemas de rutas más cortas es la extensión llamada *ArcView Network Analyst* . Esta extensión permite a usuarios solucionar una variedad de problemas usando las redes geográficas mencionadas anteriormente , por ejemplo encontrar la ruta más eficiente del recorrido o tour, la generación de direcciones del recorrido, encontrar la unidad de servicio más cercana, o definir las áreas de servicio basadas en el tiempo del recorrido. El analista de la red de ArcView cuenta con un interfaz gráfico que nos permite modelar:

- **Rutas más cortas entre cualquier par de puntos**: por ejemplo cuál de las 32 ciudades es la más cercana al Distrito Federal
- **Ruta óptima entre muchos puntos**: cuál es la ruta óptima entre cualquier par de ciudades de la república mexicana.
- **Encontrar un centro de distribución más cercano con respecto a un punto en específico**: que supermercado está más cercano a mi casa y cómo consigo llegar allí.
- **Manejo de análisis del tiempo**: cuántos son los clientes potenciales alrededor de un restaurante de comida rápida en tres, cinco y 10 minutos.

La extensión Network Analyst puede hacer un ruteo punto a punto y poner ciertos señalamientos locales en el camino. Los datos de la red geográfica se pueden basar en dibujos realizados en otros programas como CAD.

Esta extensión también incluye un conjunto de herramientas de análisis de redes muy avanzadas que nos permiten realizar un análisis de redes muy sofisticado. El algoritmo que utiliza esta extensión para resolver los problemas de redes es el algoritmo Dijkstra mejorado, el cual consiste en encontrar rutas más cortas entre cualquier par de nodos

5. Determinar el **tipo de solución:** como queremos resolver un problema del agente viajero con 32 nodos es necesario resolverlo con algún software en especial que nos muestre nuestro viaje óptimo o bien con una solución muy cercana al óptimo. Para resolver este problema es suficiente con el apoyo de las técnicas heurísticas, pero dado que existe software que nos puede dar un resultado óptimo utilizando técnicas exactas utilizaré técnicas de soluciones exactas y heurísticas.

4.4.1 Resultados del ejemplo del tour a través de las 32 ciudades de la República Mexicana utilizando WINQSB

Una vez conociendo nuestro problema procedemos a resolverlo con la ayuda de WINQSB, que debe estar previamente instalado en la computadora en donde vayas a resolver el problema.

Características de la computadora en donde se realizó la corrida del programa.

Intel Pentium III	Sistema
598 MHz	Microsoft Windows XP Professional
192 MB de RAM	Versión 2002

Tabla 4.4Características de la computadora

Una vez que se tiene instalado este programa nos vamos al ícono inicio, programas y seleccionamos el módulo Network Modeling, como se muestra en la figura de abajo.

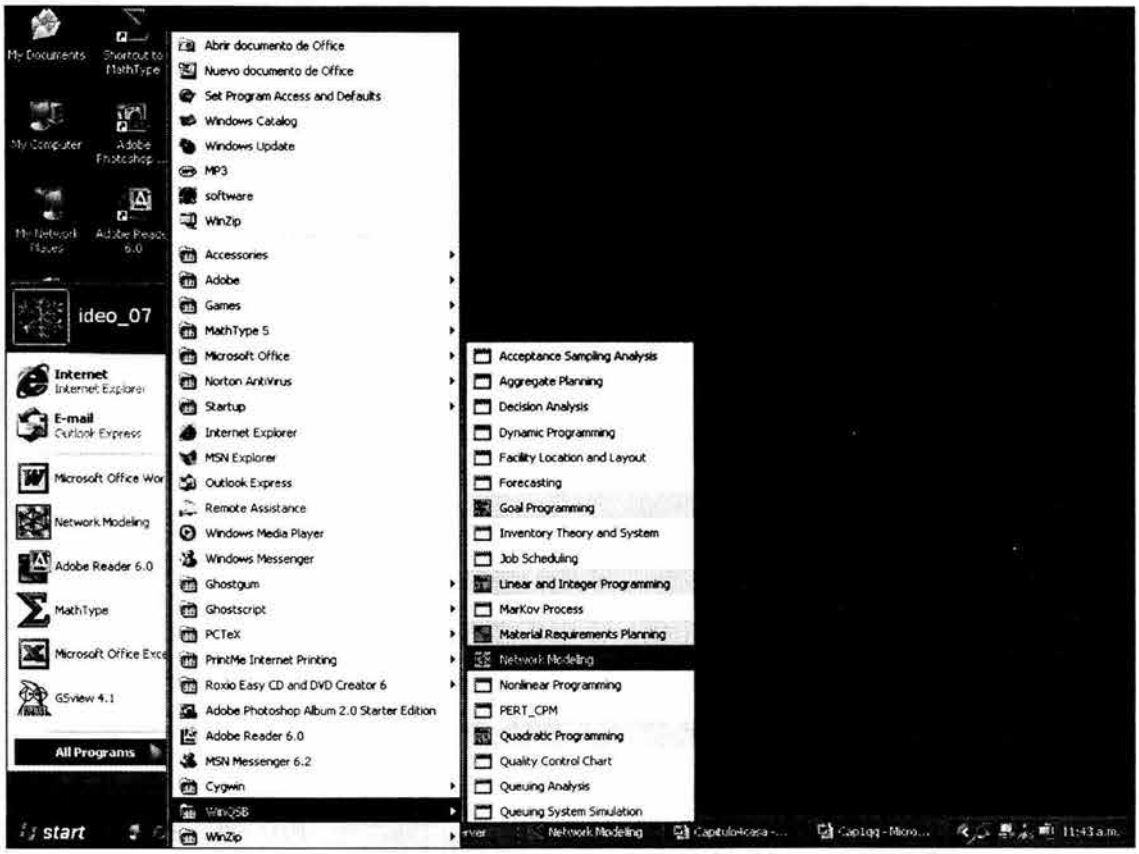


Figura 4.14 Inicio del programa WINQSB, Network Modeling

1. Para usar este programa es necesario tener los datos que se tienen que ingresar en el programa, en este caso es una matriz de adyacencia que nos representan las distancias en Km. entre cada una de las 31 ciudades y el Distrito Federal. Que se encuentra en las secciones siguientes.
2. Una vez que ya estás en programa se tiene que seleccionar el comando New Problem, con el icono . Tendrás que seleccionar el tipo de problema y especificar la información relacionada con el problema. En este caso tenemos que seleccionar "The traveling salesman problem", minimización, simétrico y poner el título del problema, así como el número de nodos que es 32. Y dar OK. Y se aparecerá una hoja de cálculo de 32 por 32.

3. En el caso de que hubiera nodos sin conectarse se tiene que poner una "M" que nos representa un costo infinito o bien dejar las celdas vacías. En este caso las ciudades que van desde la ciudad i a la ciudad i se tienen que dejar en blanco.
4. Puedes usar las teclas de navegación o la barra espaciadora para moverte en la hoja de cálculo.
5. Se tiene que dar click o doble click en la celda para seleccionarla. Y puedes introducir los datos.
6. (Opcional) Use el comando Format para cambiar el formato de los números, el tamaño de la fuente, el color, alineación, altura de los renglones y el ancho de las columnas.
7. (Opcional) Use los comandos de Edit para cambiar el nombre del problema, de los nodos, tipo de problema, criterio de la función objetivo, o bien para borrar o agregar nodos.
8. (Opcional, pero importante) Después de introducir los datos del problema, seleccionar el comando Save Problem As para salvar el problema.

Una vez realizado los pasos anteriores se presentan las siguientes pantallas:

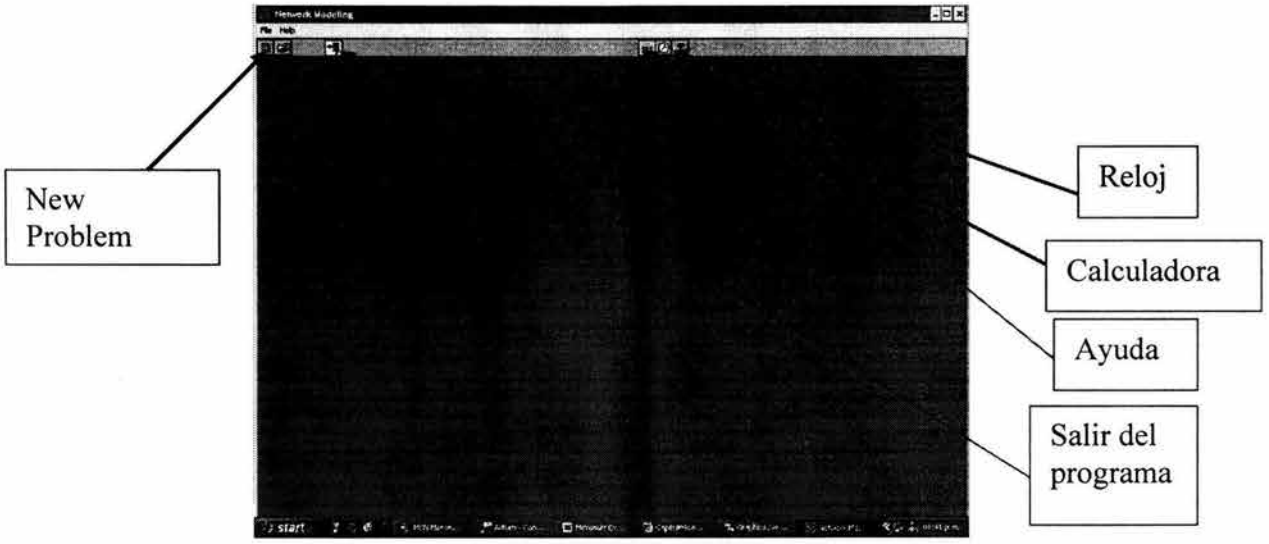
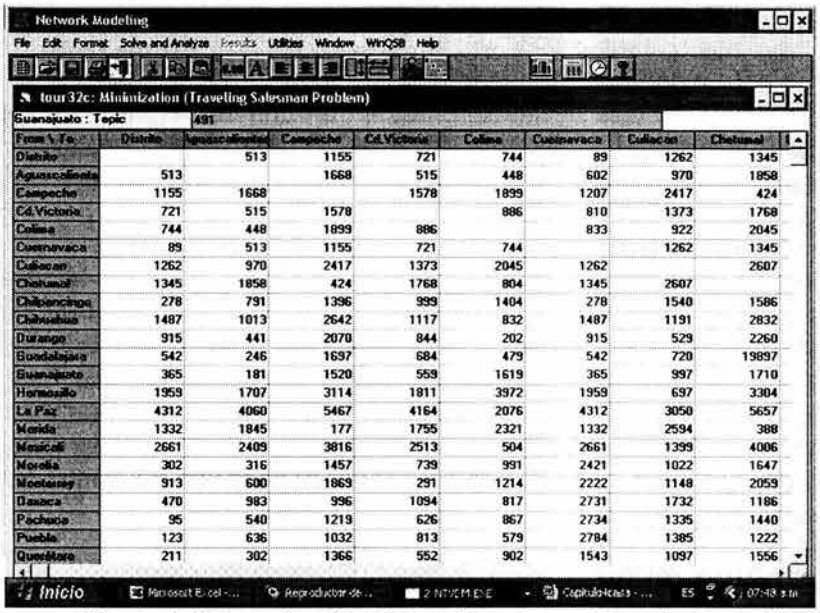


Figura 4.15 Primer hoja de WINQSB

Una vez que seleccionamos el icono New Problem se nos pregunta qué tipo de problema queremos resolver, entonces elegimos la opción "The traveling salesman problem", seleccionamos minimización, y la opción de simétrico. Tendrás que ponerle título al problema y el número de nodos que son 32. Ya que tenemos esos datos aparecerá una hoja de cálculo en la que tendrás que ingresar las distancias de ciudad entre ciudad y dejando espacios en blanco en las celdas que tienen el mismo subíndice.
Como la matriz de costos es muy grande, ésta se pueda ver al final de este capítulo.
Una vez que nos aparece la hoja de cálculo e introducimos los datos, se pueden editar los nombres de los nodos, dando click en el comando Edit y eligiendo editar nombre de los nodos. Se podrá poner el nombre de los estados de la república.



From \ To	Distrito	Aguascalientes	Campeche	Cd.Victoria	Colima	Cuernavaca	Culiacan	Chetumal
Distrito		513	1155	721	744	89	1262	1345
Aguascalientes	513		1668	515	448	602	970	1858
Campeche	1155	1668		1578	1899	1207	2417	424
Cd.Victoria	721	515	1578		886	810	1373	1768
Colima	744	448	1899	886		833	922	2045
Cuernavaca	89	513	1155	721	744		1262	1345
Culiacan	1262	970	2417	1373	2045	1262		2607
Chetumal	1345	1858	424	1768	804	1345	2607	
Chilpancingo	278	791	1396	999	1404	278	1540	1586
Chihuahua	1487	1013	2642	1117	832	1487	1191	2832
Durango	915	441	2070	844	202	915	529	2260
Guadalajara	542	246	1697	684	479	542	720	19897
Guanajuato	365	181	1520	559	1619	365	997	1710
Hermosillo	1959	1707	3114	1811	3972	1959	697	3304
La Paz	4312	4060	5467	4164	2076	4312	3050	5657
Merida	1332	1845	177	1755	2321	1332	2594	388
Mexicali	2661	2409	3816	2513	504	2661	1399	4006
Morelia	302	316	1457	739	991	2421	1022	1647
Monterrey	913	600	1869	291	1214	2222	1148	2059
Oaxaca	470	983	996	1094	817	2731	1732	1186
Pachuca	95	540	1219	626	867	2734	1335	1440
Puebla	123	636	1032	813	579	2784	1385	1222
Queretaro	211	302	1366	552	902	1543	1097	1556

Figura 4.16 Introducción de los datos y nombres de los nodos

Cuando ya se ingresaron todos los datos, damos click en solve and analyze y enseguida elegimos solve the problem, y se desplegará una imagen como se muestra a continuación:

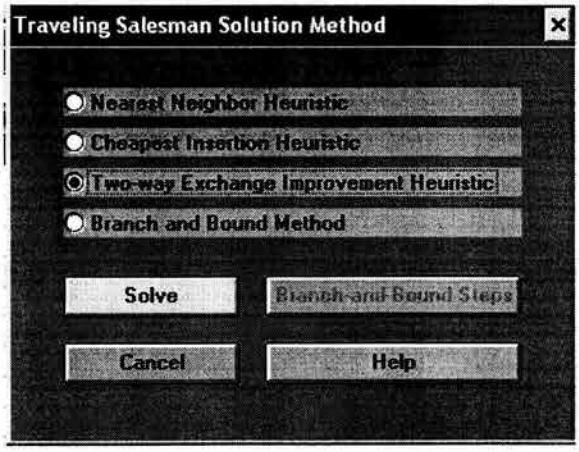


Figura 4.17 Despliegue de técnicas heurísticas y exactas del WINQSB

Se podrá notar que el programa nos facilita cuatro formas de resolver el problema, Nearest Neighbor Heuristic o vecino mas cercano, Cheapest Insertion Heuristic o inserción más barata, Two-way Exchange improvement o dos-optimal , estas técnicas son heurísticas, es decir, encuentran una solución buena pero no la óptima y la última estrategia es **Branch and Bound** que es una técnica exacta, es decir, encuentra el óptimo pero con un tiempo de computación muy costoso. La descripción de esta técnica se puede ver a detalle en el capítulo dos de esta tesis.
A continuación describo brevemente las heurísticas usadas por el programa para que el lector se dé una idea de cómo trabajan los algoritmos, aunque una descripción más detallada se puede encontrar en el capítulo 3.

Vecino más cercano: Como se comentaba en el capítulo 3, esta heurística es uno de los procedimientos heurísticos más sencillos para el TSP, que trata de construir un ciclo

Hamiltoniano de bajo costo, basándose en el vértice cercano a uno dado. Para ver más detalles respecto de este algoritmo ver el capítulo 3.
En este caso se resolvió para que Aguascalientes en un primer ejercicio y el Distrito Federal en un segundo ejercicio fueran las ciudades iniciales. En este caso se pondrán únicamente los resultados partiendo del Distrito Federal. Y se pondrán al final de la explicación de las técnicas que utiliza el programa WINQSB.

Inserción más barata: Este tipo de heurística pertenece a un grupo de técnicas que se les conoce como técnicas de inserción. Que consisten en comenzar construyendo ciclos que visiten únicamente unos cuantos vértices, para posteriormente extenderlos insertando los vértices restantes. En cada paso se inserta un nuevo vértice en el ciclo hasta obtener un ciclo hamiltoniano. Dentro de este conjunto de técnicas podemos encontrar distintas técnicas de inserción, dentro de las cuales figura el de inserción más barata que consiste en seleccionar el vértice *j* que será insertado con el menor incremento del costo. En el capítulo 3 se ve con más detalle esta técnica.

2-OPT: Este procedimiento se basa en la siguiente observación para grafos con distancias euclídeas (o en general con costos cumpliendo la desigualdad triangular). Un movimiento 2-opt consiste en eliminar dos aristas y reconectar los dos caminos resultantes de una manera diferente para obtener un nuevo ciclo. El ciclo final es más corto que el inicial.
En la figura 4.18 se despliega el resultado obtenido por la técnica del vecino más cercano.

: Minimization (Traveling Salesman Problem)

07-08-2004	From Node	Connect To	Distance/Cost		From Node	Connect To	Distance/Cost
1	Distrito	Toluca	46	17	Colima	Guadalajara	182
2	Toluca	Pachuca	141	18	Guadalajara	Mexicali	189
3	Pachuca	Tlaxcala	134	19	Mexicali	Hermosillo	682
4	Tlaxcala	Puebla	13	20	Hermosillo	Chihuahua	674
5	Puebla	Xalapa	183	21	Chihuahua	Saltillo	717
6	Xalapa	Oaxaca	360	22	Saltillo	Monterrey	69
7	Oaxaca	Tuxtla Gtz.	520	23	Monterrey	Cd. Victoria	271
8	Tuxtla Gtz.	Villahermosa	264	24	Cd. Victoria	Queretaro	532
9	Villahermosa	Campeche	367	25	Queretaro	Guanajuato	134
10	Campeche	Merida	157	26	Guanajuato	Morelia	160
11	Merida	Chetumal	368	27	Morelia	Tepic	496
12	Chetumal	S.L.Potosí	740	28	Tepic	Culiacan	486
13	S.L.Potosí	guascaliente	151	29	Culiacan	Cuernavaca	1242
14	guascaliente	Zacatecas	111	30	Cuernavaca	Chilpancingo	258
15	Zacatecas	Durango	290	31	Chilpancingo	La paz	4570
16	Durango	Colima	182	32	La paz	Distrito	4292
	Total	Minimal	Traveling	Distance	or Cost	=	18981
	(Result	from	Nearest	Neighbor	Heuristic)		

Figura 4.18 Ruta dada por el algoritmo del vecino mas cercano

La ruta propuesta por la técnica del vecino más cercano: Distrito Federal-Toluca-Pachuca-Tlaxcala-Puebla-Xalapa-Oaxaca-Tuxtla Gutiérrez-Villahermosa-Campeche-Mérida-Chetumal-San Luis Potosí-Aguscalientes-Zacatecas-Durango-Colima-Guadalajara-Mexicali-Hermosillo-Chihuahua-Saltillo-Monterrey-Cd. Victoria-Querétaro-Guanajuato-Morelia-Tepic-Culiacán-Cuernavaca-Chilpancingo-La Paz-Distrito Federal. Si tomamos esta ruta tendríamos que recorrer **18981** Km, a través de las 32 ciudades de la República Mexicana.

Si observamos la ruta tenemos que los últimos estados que tendríamos que visitar están muy lejanos Chilpancingo-La Paz-Distrito Federal, aquí se observa la ineficiencia de este algoritmo, ya que en un principio, sí toma las ciudades más cercanas pero, una vez que se agotan las posibilidades hasta el final se corre el riesgo de viajar entre ciudades lejanas. Es

por esto que a este tipo de algoritmos se les conoce como algoritmos miopes porque no ven más allá de la solución inmediata.

El siguiente resultado es la que se obtiene mediante la técnica inserción más barata ó Cheapest insertion heuristic.

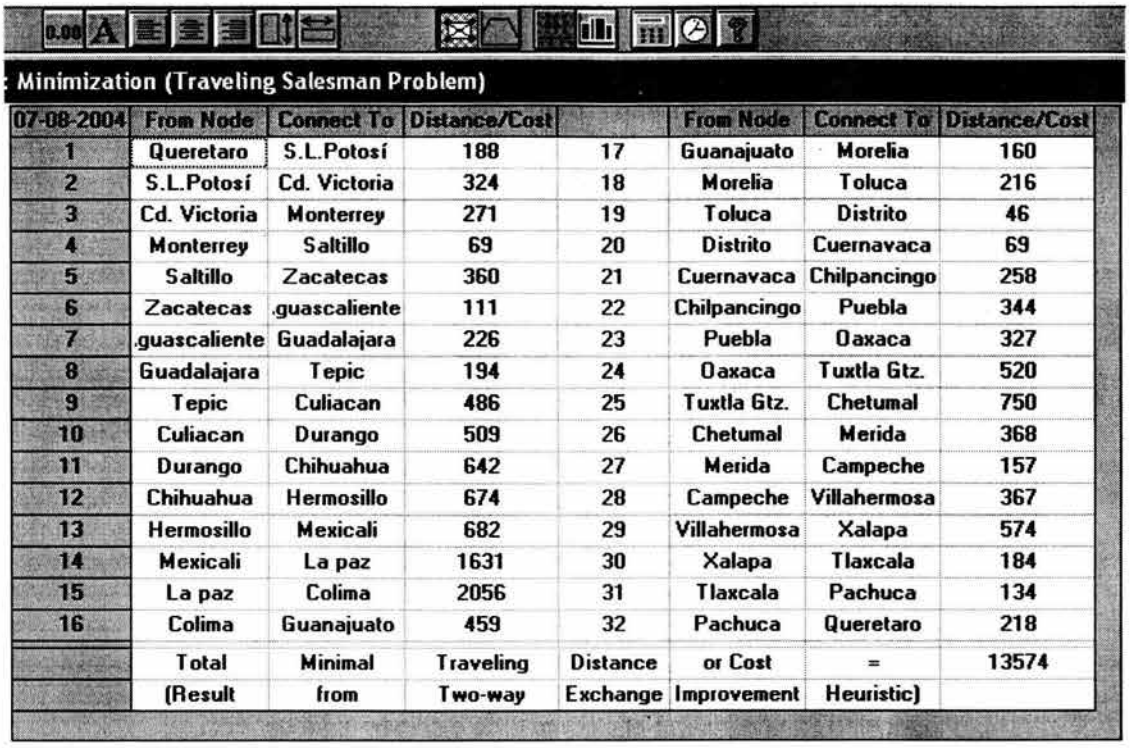
n: Minimization (Traveling Salesman Problem)

07-08-2004	From Node	Connect To	Distance/Cost		From Node	Connect To	Distance/Cost
1	Puebla	Chilpancingo	344	17	Tepic	Guadalajara	194
2	Chilpancingo	Queretaro	469	18	Guadalajara	Cuernavaca	522
3	Queretaro	S.L.Potosí	188	19	Cuernavaca	Guanajuato	345
4	S.L.Potosí	Zacatecas	170	20	Guanajuato	Morelia	160
5	Zacatecas	guascaliente	111	21	Morelia	Toluca	216
6	guascaliente	Cd. Victoria	495	22	Toluca	Distrito	46
7	Cd. Victoria	Monterrey	271	23	Distrito	Pachuca	75
8	Monterrey	Saltillo	69	24	Pachuca	Tlaxcala	134
9	Saltillo	Chihuahua	717	25	Tlaxcala	Xalapa	184
10	Chihuahua	La paz	3027	26	Xalapa	Villahermosa	574
11	La paz	Mexicali	1631	27	Villahermosa	Chetumal	557
12	Mexicali	Hermosillo	682	28	Chetumal	Merida	368
13	Hermosillo	Durango	1206	29	Merida	Campeche	157
14	Durango	Colima	182	30	Campeche	Tuxtla Gtz.	632
15	Colima	Culiacan	902	31	Tuxtla Gtz.	Oaxaca	520
16	Culiacan	Tepic	486	32	Oaxaca	Puebla	327
	Total	Minimal	Traveling	Distance	or Cost	=	15961
	(Result	from	Cheapest	Insertion	Heuristic)		

Figura 4.19 Ruta dada por el algoritmo de inserción más barata

El tour propuesto es el siguiente: Puebla-Chilpancingo-Querétaro-San Luis Potosí-Zacatecas-Aguscalientes-Cd. Victoria-Monterrey-Saltillo-Chihuahua-La Paz-Mexicali-Hermosillo-Durango-Colima-Culiacan-Tepic-Guadalajara-Cuernavaca-Guanajuato-Morelia-Toluca-Distrito Federal-Pachuca-Tlaxcala-Xalapa-Villahermosa-Chetumal-Merida-Campeche-Tuxtla Gutiérrez-Oaxaca-Puebla. Observamos que difiere con la solución anterior ya que en ésta técnica de lo que se trata es ir tomando del conjunto de ciudades mas cercanas tomar la más cercana al circuito que ya se tiene formado previamente. Por consiguiente las ciudades que toma son siempre las que nos garantizan un circuito de costo mínimo. Si tomaramos esta ruta tendríamos que recorrer **15961** Km a través de las 32 ciudades.

Se presenta a continuación la solución propuesta por el algoritmo 2-OPT ó Two-way Exchange Improvement Heuristic.



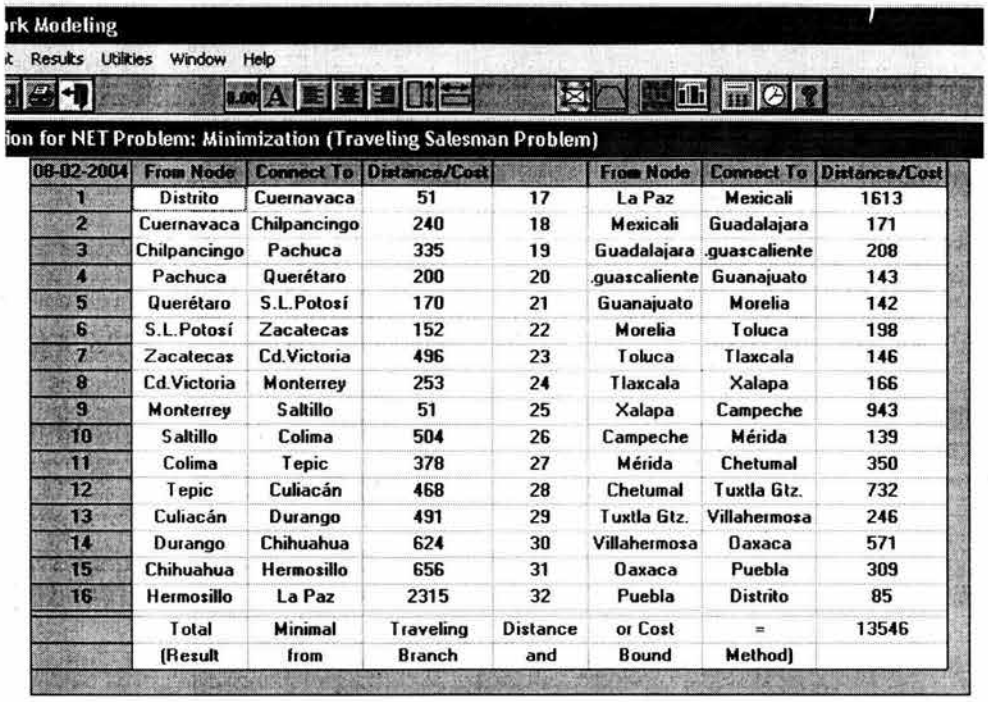
Minimization (Traveling Salesman Problem)

07-08-2004	From Node	Connect To	Distance/Cost		From Node	Connect To	Distance/Cost
1	Queretaro	S.L.Potosí	188	17	Guanajuato	Morelia	160
2	S.L.Potosí	Cd. Victoria	324	18	Morelia	Toluca	216
3	Cd. Victoria	Monterrey	271	19	Toluca	Distrito	46
4	Monterrey	Saltillo	69	20	Distrito	Cuernavaca	69
5	Saltillo	Zacatecas	360	21	Cuernavaca	Chilpancingo	258
6	Zacatecas	guascaliente	111	22	Chilpancingo	Puebla	344
7	guascaliente	Guadalajara	226	23	Puebla	Oaxaca	327
8	Guadalajara	Tepic	194	24	Oaxaca	Tuxtla Gtz.	520
9	Tepic	Culiacan	486	25	Tuxtla Gtz.	Chetumal	750
10	Culiacan	Durango	509	26	Chetumal	Merida	368
11	Durango	Chihuahua	642	27	Merida	Campeche	157
12	Chihuahua	Hermosillo	674	28	Campeche	Villahermosa	367
13	Hermosillo	Mexicali	682	29	Villahermosa	Xalapa	574
14	Mexicali	La paz	1631	30	Xalapa	Tlaxcala	184
15	La paz	Colima	2056	31	Tlaxcala	Pachuca	134
16	Colima	Guanajuato	459	32	Pachuca	Queretaro	218
	Total	Minimal	Traveling	Distance	or Cost	=	13574
	(Result	from	Two-way	Exchange	Improvement	Heuristic)	

Figura 4.20 Ruta dada por el algoritmo de 2-OPT

La ruta propuesta por este algoritmo es: Querétaro-San Luis Potosí-Cd. Victoria-Monterrey-Saltillo-Zacatecas-Aguascalientes-Guadalajara-Tepic-Culiacán-Durango-Chuihuahua-Hermosillo-Mexicali-La Paz-Colima-Guanajuato-Morelia-Toluca-Distrito Federal-Cuernavaca-Chilpancingo-Puebla-Oaxaca-Tuxtla-Chetumal-Merida-Campeche-Villahermosa-Xalapa-Tlaxcala-Pachuca-Querrétaro recorriendo **13,574** Km., con lo cual se observa que esta técnica es la que nos proporciona un mayor ahorro en el recorrido con respecto las técnicas heurísticas anteriores.

El siguiente resultado es el óptimo con una función de objetivo de **13, 546,** ya que es el que se obtiene por la técnica de ramificación y acotamiento o Branch and Bound. Lo cual quiere decir que el vacacionista tendría que viajar 13, 548 Km y visitaría cada una de las 32 ciudades principales de la República Mexicana. Cabe señalar que este resultado se obtiene a través de un esfuerzo computacional arduo, la respuesta se obtuvo con 13 días y dos horas 19 min de corrida del programa WINQSB, con lo que se observa que tenemos la ruta óptima, pero a cambio de esperar un tiempo para saber la respuesta y en este caso sí queremos saber la respuesta de manera inmediata, ya que como comenté en un principio ¡se nos terminan las vacaciones! Es por esto que conviene tomar el resultado propuesto por la técnica heurística 2-Opt,ya que se ha demostrado que es una solución bastante cercana al óptimo y la solución se obtiene de manera inmediata, aunque en este caso como se conoce la ruta óptima, pues se toma la solución óptima. En la figura 4.20 se muestra el resultado óptimo obtenido por el programa WINQSB mediante la técnica Branch and Bound.



rk Modeling

Results Utilities Window Help

ion for NET Problem: Minimization (Traveling Salesman Problem)

08-02-2004	From Node	Connect To	Distance/Cost		From Node	Connect To	Distance/Cost
1	Distrito	Cuernavaca	51	17	La Paz	Mexicali	1613
2	Cuernavaca	Chilpancingo	240	18	Mexicali	Guadalajara	171
3	Chilpancingo	Pachuca	335	19	Guadalajara	.guascaliente	208
4	Pachuca	Querétaro	200	20	.guascaliente	Guanajuato	143
5	Querétaro	S.L.Potosí	170	21	Guanajuato	Morelia	142
6	S.L.Potosí	Zacatecas	152	22	Morelia	Toluca	198
7	Zacatecas	Cd.Victoria	496	23	Toluca	Tlaxcala	146
8	Cd.Victoria	Monterrey	253	24	Tlaxcala	Xalapa	166
9	Monterrey	Saltillo	51	25	Xalapa	Campeche	943
10	Saltillo	Colima	504	26	Campeche	Mérida	139
11	Colima	Tepic	378	27	Mérida	Chetumal	350
12	Tepic	Culiacán	468	28	Chetumal	Tuxtla Gtz.	732
13	Culiacán	Durango	491	29	Tuxtla Gtz.	Villahermosa	246
14	Durango	Chihuahua	624	30	Villahermosa	Oaxaca	571
15	Chihuahua	Hermosillo	656	31	Oaxaca	Puebla	309
16	Hermosillo	La Paz	2315	32	Puebla	Distrito	85
	Total	Minimal	Traveling	Distance	or Cost	=	13546
	(Result	from	Branch	and	Bound	Method)	

Figura 4.21 Resultado óptimo obtenido por la técnica Branch and Bound

4.4.2 Análisis de resultados del ejemplo del tour a través de las 32 ciudades de la República Mexicana con WINQSB

Aunque normalmente se recurre al algoritmo aproximado por no existir un método exacto para obtener el óptimo, o por ser éste computacionalmente muy costoso, en ocasiones puede que dispongamos de un procedimiento que proporcione el óptimo para un conjunto limitado de ejemplos (usualmente de tamaño reducido). Este conjunto de ejemplos puede servir para medir la calidad del método heurístico.

Normalmente se mide, para cada uno de los ejemplos, la desviación porcentual de la solución heurística frente a la óptima, calculando posteriormente el promedio de dichas desviaciones. Si llamamos c_h al costo de la solución del algoritmo heurístico y c_{opt} al costo de la solución óptima de un ejemplo dado, en un problema de minimización la desviación porcentual viene dada por la expresión: $\frac{c_h - c_{opt}}{c_{opt}} \cdot 100$.

En la tabla 4.7 se muestran los resultados de las técnicas que se utilizaron, se puede observar que la técnica heurística del vecino más cercano es una técnica que es muy fácil de realizar, pero es la que más se aleja de la solución óptima y por otro lado la técnica 2-Opt es bastante eficiente, de hecho, casi es una solución óptima, de ahí que la ocupen de manera frecuente en los estudios comparativos.

Tipo de técnica	Tiempo	Desviación del óptimo (%)	Función objetivo
Vecino más cercano	1 seg	$C_{NN} = \frac{18981-13512}{13512} = \frac{5469}{13512} = 40$	18981 Km
Inserción más barata	1 seg	$C_{CH} = \frac{15961-13512}{13512} = .181 \approx 18$	15961 Km
2-OPT	1 seg	$C_{2OPT} = \frac{13574-13512}{13512} = .45$	13574 Km
Ramificación y acotamiento	13 días	----------	13512 Km

Tabla 4.7 Resumen de resultados del ejemplo

Pues bien, ya con maleta en mano, el vacacionista debe tomar la siguiente ruta para optimizar las distancias, tiempo y gastos. Esta ruta es la propuesta por Branch and Bound.

Distrito Federal, Cuernavaca, Chilpancingo, Pachuca, Querétaro, S.L. Potosí, Zacatecas, Cd. Victoria, Monterrey, Saltillo, Colima, Tepic, Culiacán, Durango, Chihuahua, Hermosillo, la Paz, Mexicali, Guadalajara, Aguascalientes, Guanajuato, Morelia, Toluca, Tlaxcala, Xalapa, Campeche, Mérida, Chetumal, Tuxtla Gtz, Villahermosa, Oaxaca, Puebla, Distrito Federal.

4.4.3 Solución utilizando el software ARCVIEW 3.2.

Para resolver este ejemplo con el programa ARCVIEW es necesario contar con este programa previamente instalado en una computadora, en este caso, este software se encuentra instalado en el laboratorio de transporte y con el apoyo de Ing. Raúl Espinoza Jiménez, quien es especialista en este software, se realizó la corrida del ejemplo .

Las características de la computadora en donde se realizó esta corrida son las siguientes:

Intel Pentium IV	Sistema
2.26 GHz	Microsoft Windows XP Professional
512 MB de RAM	Versión 2002

Tabla 4.8 Características de la computadora

Dado que el programa ya cuenta con las distancias entre cada par de ciudades no es necesario introducir estos datos, y al momento de resolverlo activamos las opciones de encontrar la ruta más corta entre cualquier par de puntos y también activamos la opción de regresar a la ciudad de origen, conjuntando ambas restricciones estamos resolviendo el problema del agente viajero. Cabe señalar que el programa cuenta con levantamientos de datos de 5 tipos de carreteras: autopistas, supercarreteras, carreteras tipo A, B y C.

Para resolver este ejemplo se restringió el programa para que calculara el tour considerando únicamente las autopistas, dado que en el programa WINQSB los datos que se introdujeron son tomados del guía roji y son distancias entre ciudades por autopistas y para tener igualdad de circunstancias se consideraron únicamente autopistas. En cuestión de segundos el software nos arrojó los siguientes datos:

PATH_I	DF_LABEL	T_LABEL	F_COST	T_COST	CIUDAD
1	Site #6324	Site #6431	0	0.518	Ciudad de México
2	Site #6431	Site #6143	0.518	2.478	Toluca
3	Site #6143	Site #6471	2.478	5.997	Morelia
4	Site #6471	Site #5045	5.997	9.57	Colima
5	Site #5045	Site #5423	9.57	11.52	Tepic
6	Site #5423	Site #4909	11.52	13.649	Guadalajara
7	Site #4909	Site #4648	13.649	14.702	Aguascalientes
8	Site #4648	Site #4338	14.702	17.474	Zacatecas
9	Site #4338	Site #3924	17.474	20.998	Durango
10	Site #3924	Site #1099	20.998	27.657	Culiacán
11	Site #1099	Site #4289	27.657	46.223	Hermosillo
12	Site #4289	Site #1	46.223	58.602	La paz
13	Site #1	Site #1437	58.602	71.029	Mexicali
14	Site #1437	Site #3567	71.029	77.817	Chihuahua
15	Site #3567	Site #3348	77.817	78.681	Saltillo
16	Site #3348	Site #4443	78.681	79.001	Monterrey
17	Site #4443	Site #4448	79.001	81.351	Ciudad victoria
18	Site #4448	Site #4808	81.351	84.343	San Luis Potosí
19	Site #4808	Site #5236	84.343	86.124	Guanajuato
20	Site #5236	Site #5479	86.124	87.36	Querétaro
21	Site #5479	Site #5826	87.36	89.324	Pachuca
22	Site #5826	Site #6415	89.324	90.564	Tlaxcala
23	Site #6415	Site #6565	90.564	90.841	Puebla
24	Site #6565	Site #6261	90.841	92.514	Jalapa
25	Site #6261	Site #7123	92.514	101.607	Villahermosa
26	Site #7123	Site #6870	101.607	106.999	Campeche
27	Site #6870	Site #5247	106.999	110.317	Mérida
28	Site #5247	Site #6038	110.317	113.022	Chetumal
29	Site #6038	Site #7451	113.022	118.333	Tuxtla Gutiérrez
30	Site #7451	Site #7348	118.333	122.786	Oaxaca
31	Site #7348	Site #7270	122.786	126.664	Chilpancingo
32	Site #7270	Site #6655	126.664	128.285	Cuernavaca
33	Site #6324	Site #6324	128.285	128.902	Ciudad de México

Tabla 4.9 Ruta (óptima) obtenida con ARCVIEW 3.2

Con una función objetivo de **13521.14** Km. Lo cual quiere decir que el agente viajero o el vacacionista debe hacer un recorrido total de 13521 Km, visitando las 32 ciudades de la república mexicana, partiendo del distrito federal y regresando a este mismo punto. El orden en que deben ser visitadas las ciudades se refleja en la última columna de la tabla 4.7. Este tour es el que debe realizar el vacacionista para que la distancia recorrida sea la mínima. Y dado que el programa utiliza un algoritmo exacto para resolver el problema la ruta que se obtuvo es la ruta óptima.

En la figura 4.22 se muestra la ruta óptima arrojada por el programa arcview 3.2.

El punto de color amarillo es donde se encuentra localizado el Distrito Federal que es de donde estamos partiendo y el lugar a donde tenemos que regresar.

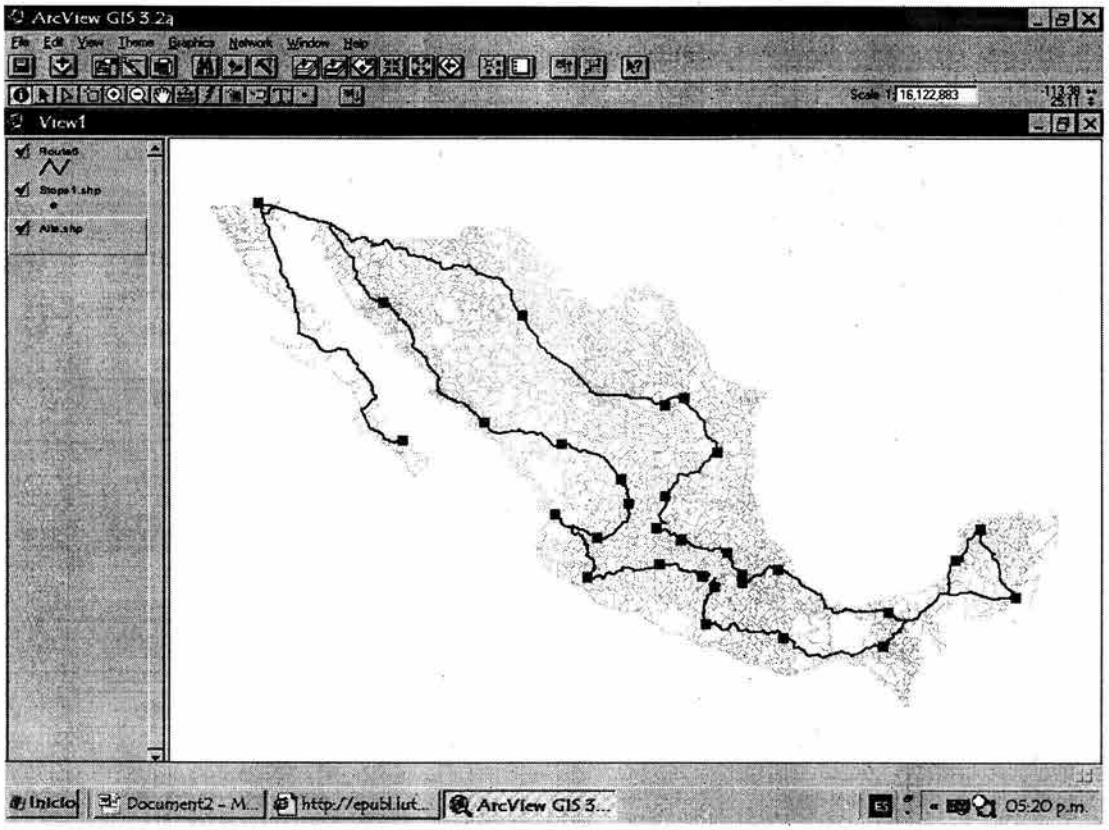


Figura 4.22 Tour óptimo a través de las 32 ciudades de la República Mexicana con ARCVIEW3.2

4.4.3 Comparación de resultados de WINQSB con ARCVIEW 3.2

Se observa que la técnica que nos proporciona la solución óptima del software de WINQSB es la técnica Branch and Bound, aunque se observa también que la técnica 2-OPT es una técnica bastante eficiente que tiene una desviación con respecto del óptimo en .4%, es decir, es una técnica bastante eficiente (en tiempo y exactitud) y la solución proporcionada por el programa AERCVIEW 3.2 es una solución óptima, dado que el algoritmo que resuelve el problema de la ruta más corta es un algoritmo de solución exacta (Dijkstra mejorado, en el anexo se explica el funcionamiento de este algoritmo) se hará una comparación de estas tres técnicas para determinar la diferencia en la función objetivo y las rutas propuestas por cada una de estas técnicas.
En la tabla 4.10 se muestran las cotas que se obtuvieron por cada una de las técnicas descritas con anterioridad:

Software utilizado	Técnica utilizada	Tiempo	Función objetivo	Desviación con el óptimo (%) de la técnica 2-OPT
WINQSB	2-OPT	1 seg	13,574 Km	
WINQSB	Branch and Bound (B&B)	13 dias, 2 horas 19 min.	13,512 Km	$C_{2-OPT} = \left(\frac{13574-13512}{13512}\right)*(100) = 0.45$
ARCVIEW 3.2	Dijkstra mejorado	1 seg	13, 521Km	$C_{2-OPT} = \left(\frac{13574-13521}{13521}\right)*(100) \approx .40$

Tabla 4.10 Desviación con respecto a los óptimos B&B y Dijkstra mejorado de la técnica 2-OPT

Se puede observar que las soluciones son muy cercanas aun cuando la técnica *2-OPT* sea una técnica heurística, se muestra la gran eficiencia que tiene con respecto a las dos soluciones que se tienen que son técnicas exactas, lo que nos hace recomendar este tipo de técnica, ya que el resultado que tenemos es muy cercano al óptimo y con una velocidad de respuesta increíble. También se hace notar que el tiempo de respuesta que tenemos por el programa WINQSB es excesivamente lejano con respecto la velocidad de respuesta que obtenemos con el programa ARCVIEW 3.2.

4.5 Algoritmos desarrollados hasta el momento para resolver el problema del agente viajero y sus variantes

En los últimos veinte años ha habido un gran desarrollo en las técnicas heurísticas y de manera muy especial en las técnicas metaheurísticas, en la mayoría de los nuevos algoritmos que se han desarrollado hay variaciones en las estrategias de la búsqueda local, es decir, como se eligen los conjuntos de soluciones a los que le llaman vecindades.

También se han desarrollado algoritmos que se basan en la técnica $k-OPT$, la cual ha resultado ser una de las técnicas heurísticas de mejora con mayor éxito.

En la tabla 4.7 se presentan algunas de las técnicas desarrolladas hasta el momento junto con el número de nodos, cabe señalar que la tabla que se presenta a continuación fue desarrollada por la Dra. Idalia Flores de la Mota, complementándola en algunas técnicas heurísticas.

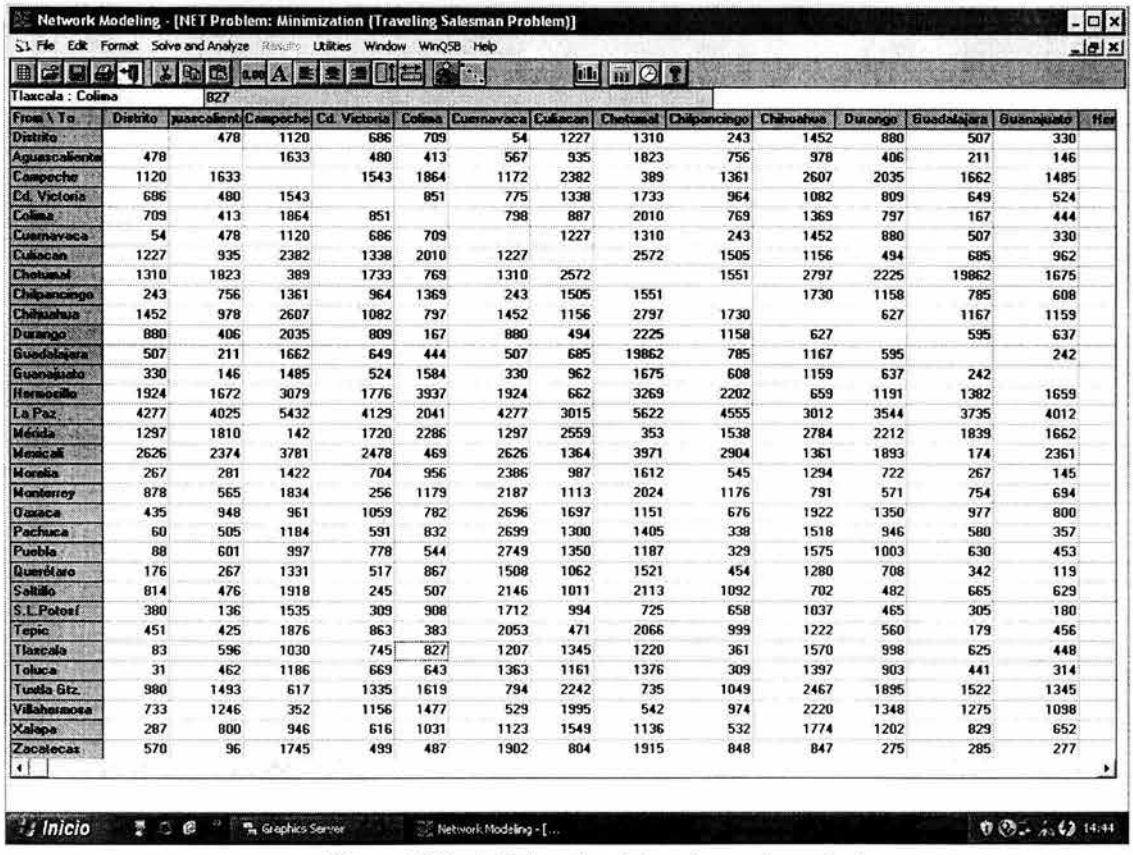
TIPO DE PAV	AUTOR	ALGORITMO	COMPLEJIDAD COMPUTACIONAL	TAMAÑO DEL PROBLEMA
SIMÉTRICO	Tsubikatani y Evans (1998)	BÚSQUEDA TABÚ RAMA Y COTA	LIMITADO POR EL TIEMPO DE CPU	50 PROBLEMAS CON 20, 50 Y 100 NODOS
ASIMÉTRICO	Balas y Fischetti (1999)	DESIGUALDADES ELEVADAS DE CICLO	$O(n^3)$	
F.O. MINIMAX	Somhom, Modares Y Enkawa (1999)	RED NEURONAL ADAPTATIVA	LIMITADO POR EL TIEMPO DE CPU	50 A 134 NODOS

BÚSQUEDA DE K- TOURS MEJORADOS	Van der Poort et. Al. (1999)	EXACTOS: PARTICIONAMIENTO Y RAMA Y COTA		50 PROBLEMAS DE 17 A 137 NODOS
EUCLIDEANO	Tassiulas (1997)	HEURÍSTICO: VECINO MÁS CERCANO	$2.482\sqrt{n}+O(\sqrt{n})$	
EUCLIDEANO CON ESTRUCTURA DE ÁRBOL P-Q	Burkard, Deineko y Woeginger (1999)	HEURÍSTICO :PROG. DINÁMICA MODIFICADO	$O(n^3 2^d)$	
GENERAL	Gutin (1999)	HEURÍSTICO: BÚSQUEDA LOCAL DE VECINDADES	$O(r^s n)$	
GENERAL	Kabadi y Baki (1999)	EXACTO: BASADO EN GILMORE-GOMORY Y ÁRBOL DE EXPANSIÓN	$O(2^{\sum(vi+1)} n^{(2l+2)})\ Sl \sum(vi+1) \le k$	
GENERAL	SHI ET. AL (1999)	HEURÍSTICO: PARALELO ALEATORIZADO, PARTICIONES ANIDADAS	LIMITADO POR EL TIEMPO DE CPU	51 NODOS Y 417 SI ES EUCLIDEANO
APLICACIÓN A CALENDARIZACIÓN DE PARTES Y ACTIVIDADES DE ROBOTS	Aneja y Kamow (1999)	EXACTO: BASADO EN ÁRBOLES DE EXPANSIÓN	$O(n \log n)$	
GENERAL	Smith (1999)	REDES NEURONALES		CITA VARIOS PROBLEMAS RESUELTOS POR OTROS AUTORES
GENERAL	Marco Dorigo (1996)	ANT COLONY SYSTEM	$O(NCn^2 m)$	5 CONJUNTOS DE 100 CIUDADES CADA UNO
GENERAL	Rosenkrantz, Stearns y Lewis (1977)	HEURÍSTICO: VECINO MÁS CERCANO	$O(n^2)$	10000
GENERAL	Stearns (1978)	HEURISTICO: ALGORITMO DE INSERCIÓN	$O(n^2 \log n)$	10000
SIMÉTRICO	Prim (1957)	HEURÍSTICO: ALGORITMO PRIM	$O(n^2)$	10000
GENERAL	Kruskal (1956)	HEURÍSTICO: ALGORITMO KRUSKAL	$O(n^2 + q \log n)$	10000
GENERAL	Christofides (1976)	HEURÍSTICO: ALGORITMO CHRISTOFIDES	$O(n^2)$	10000
GENERAL	Clarke y Wright (1964)	HEURÍSTICO: ALGORITMO DE AHORROS	$O(n^3)$	10000

GENERAL	Lin y Kernighan (1973)	HEURÍSTICO: ALGORITMO 2-OPT	$O(n^2)$	10000
GENERAL	Lin y Kernighan (1973)	HEURÍSTICO: ALGORITMO LIN Y KERNIGHAN	$O(n^2)$	10000

Tabla 4.9 Algunos algoritmos desarrollados para resolver el problema del agente viajero.

A continuación se muestran las distancias entre cada par de ciudades (vía autopista) que se utilizaron para resolver el problema del agente viajero, tomados del Guía Roji e insertados en el software WINQSB.



Network Modeling - [NET Problem: Minimization (Traveling Salesman Problem)]

File Edit Format Solve and Analyze Results Utilities Window WinQSB Help

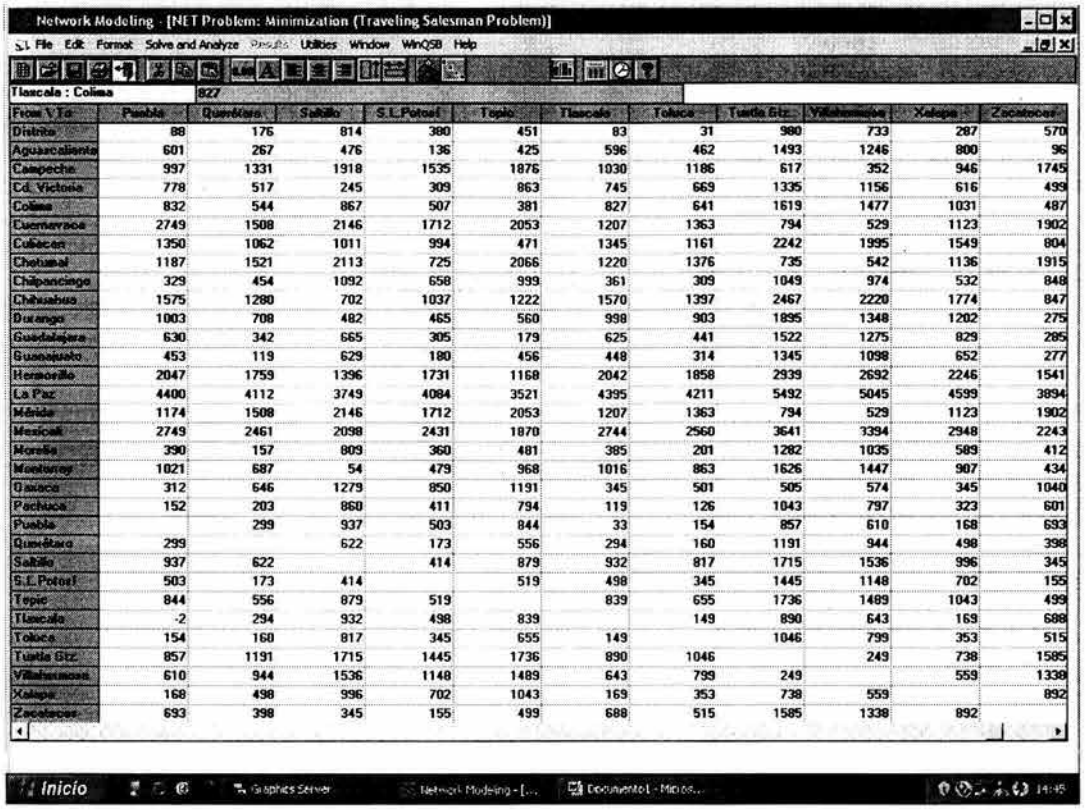
Tlaxcala : Colima 827

From \ To	Distrito	uascalient	Campeche	Cd. Victoria	Colima	Cuernavaca	Culiacan	Chetumal	Chilpancingo	Chihuahua	Durango	Guadalajara	Guanajuato	Her
Distrito		478	1120	686	709	54	1227	1310	243	1452	880	507	330	
Aguascalientes	478		1633	480	413	567	935	1823	756	978	406	211	146	
Campeche	1120	1633		1543	1864	1172	2382	389	1361	2607	2035	1662	1485	
Cd. Victoria	686	480	1543		851	775	1338	1733	964	1082	809	649	524	
Colima	709	413	1864	851		798	887	2010	769	1369	797	167	444	
Cuernavaca	54	478	1120	686	709		1227	1310	243	1452	880	507	330	
Culiacan	1227	935	2382	1338	2010	1227		2572	1505	1156	494	685	962	
Chetumal	1310	1823	389	1733	769	1310	2572		1551	2797	2225	19862	1675	
Chilpancingo	243	756	1361	964	1369	243	1505	1551		1730	1158	785	608	
Chihuahua	1452	978	2607	1082	797	1452	1156	2797	1730		627	1167	1159	
Durango	880	406	2035	809	167	880	494	2225	1158	627		595	637	
Guadalajara	507	211	1662	649	444	507	685	19862	785	1167	595		242	
Guanajuato	330	146	1485	524	1584	330	962	1675	608	1159	637	242		
Hermosillo	1924	1672	3079	1776	3937	1924	662	3269	2202	659	1191	1382	1659	
La Paz	4277	4025	5432	4129	2041	4277	3015	5622	4555	3012	3544	3735	4012	
Mérida	1297	1810	142	1720	2286	1297	2559	353	1538	2784	2212	1839	1662	
Mexicali	2626	2374	3781	2478	469	2626	1364	3971	2904	1361	1893	174	2361	
Morelia	267	281	1422	704	956	2386	987	1612	545	1294	722	267	145	
Monterrey	878	565	1834	256	1179	2187	1113	2024	1176	791	571	754	694	
Oaxaca	435	948	961	1059	782	2696	1697	1151	676	1922	1350	977	800	
Pachuca	60	505	1184	591	832	2699	1300	1405	338	1518	946	580	357	
Puebla	88	601	997	778	544	2749	1350	1187	329	1575	1003	630	453	
Querétaro	176	267	1331	517	867	1508	1062	1521	454	1280	708	342	119	
Saltillo	814	476	1918	245	507	2146	1011	2113	1092	702	482	665	629	
S.L.Potosí	380	136	1535	309	908	1712	994	725	658	1037	465	305	180	
Tepic	451	425	1876	863	383	2053	471	2066	999	1222	560	179	456	
Tlaxcala	83	596	1030	745	827	1207	1345	1220	361	1570	998	625	448	
Toluca	31	462	1186	669	643	1363	1161	1376	309	1397	903	441	314	
Tuxtla Gtz.	980	1493	617	1335	1619	794	2242	735	1049	2467	1895	1522	1345	
Villahermosa	733	1246	352	1156	1477	529	1995	542	974	2220	1348	1275	1098	
Xalapa	287	800	946	616	1031	1123	1549	1136	532	1774	1202	829	652	
Zacatecas	570	96	1745	499	487	1902	804	1915	848	847	275	285	277	

Inicio Graphics Server Network Modeling - [...

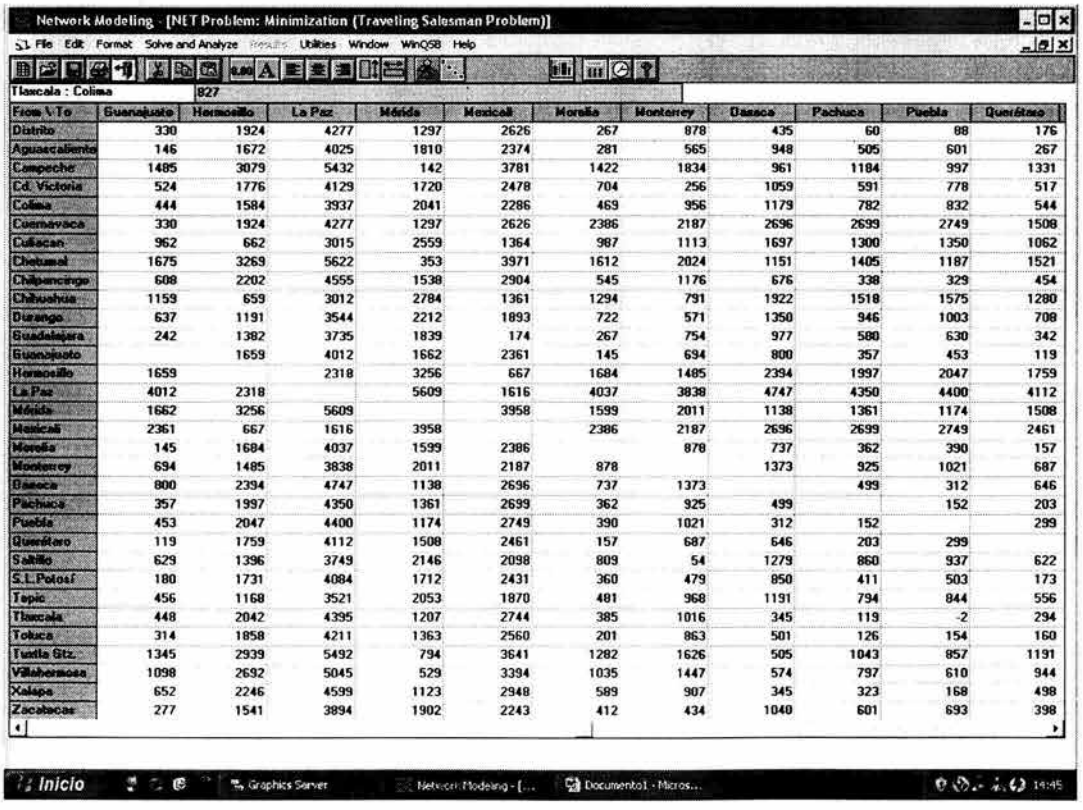
Figura 4.22 a) Datos de distancias entre estados

Dado que es una matriz muy grande (32X32), se muestran en otras dos tablas los datos de los estados faltantes.



From \ To	Puebla	Querétaro	Saltillo	S.L.Potosí	Tepic	Tlaxcala	Toluca	Tuxtla Gtz.	Villahermosa	Xalapa	Zacatecas
Distrito	88	176	814	380	451	83	31	980	733	287	570
Aguascalientes	601	267	476	136	425	596	462	1493	1246	800	96
Campeche	997	1331	1918	1535	1876	1030	1186	617	352	946	1745
Cd. Victoria	778	517	245	309	863	745	669	1335	1156	616	499
Colima	832	544	867	507	381	827	641	1619	1477	1031	487
Cuernavaca	2749	1508	2146	1712	2053	1207	1363	794	529	1123	1902
Culiacan	1350	1062	1011	994	471	1345	1161	2242	1995	1549	804
Chetumal	1187	1521	2113	725	2066	1220	1376	735	542	1136	1915
Chilpancingo	329	454	1092	658	999	361	309	1049	974	532	848
Chihuahua	1575	1280	702	1037	1222	1570	1397	2467	2220	1774	847
Durango	1003	708	482	465	560	998	903	1895	1348	1202	275
Guadalajara	630	342	665	305	179	625	441	1522	1275	829	285
Guanajuato	453	119	629	180	456	448	314	1345	1098	652	277
Hermosillo	2047	1759	1396	1731	1168	2042	1858	2939	2692	2246	1541
La Paz	4400	4112	3749	4084	3521	4395	4211	5492	5045	4599	3894
Mérida	1174	1508	2146	1712	2053	1207	1363	794	529	1123	1902
Mexicali	2749	2461	2098	2431	1870	2744	2560	3641	3394	2948	2243
Morelia	390	157	809	360	481	385	201	1282	1035	589	412
Monterrey	1021	687	54	479	968	1016	863	1626	1447	907	434
Oaxaca	312	646	1279	850	1191	345	501	505	574	345	1040
Pachuca	152	203	860	411	794	119	126	1043	797	323	601
Puebla		299	937	503	844	33	154	857	610	168	693
Querétaro	299		622	173	556	294	160	1191	944	498	398
Saltillo	937	622		414	879	932	817	1715	1536	996	345
S.L.Potosí	503	173	414		519	498	345	1445	1148	702	155
Tepic	844	556	879	519		839	655	1736	1489	1043	499
Tlaxcala	-2	294	932	498	839		149	890	643	169	688
Toluca	154	160	817	345	655	149		1046	799	353	515
Tuxtla Gtz.	857	1191	1715	1445	1736	890	1046		249	738	1585
Villahermosa	610	944	1536	1148	1489	643	799	249		559	1338
Xalapa	168	498	996	702	1043	169	353	738	559		892
Zacatecas	693	398	345	155	499	688	515	1585	1338	892	

Figura 4.22 b) Datos de distancias entre estados



From \ To	Guanajuato	Hermosillo	La Paz	Mérida	Mexicali	Morelia	Monterrey	Oaxaca	Pachuca	Puebla	Querétaro
Distrito	330	1924	4277	1297	2626	267	878	435	60	88	176
Aguascalientes	146	1672	4025	1810	2374	281	565	948	505	601	267
Campeche	1485	3079	5432	142	3781	1422	1834	961	1184	997	1331
Cd. Victoria	524	1776	4129	1720	2478	704	256	1059	591	778	517
Colima	444	1584	3937	2041	2286	469	956	1179	782	832	544
Cuernavaca	330	1924	4277	1297	2626	2386	2187	2696	2699	2749	1508
Culiacan	962	662	3015	2559	1364	987	1113	1697	1300	1350	1062
Chetumal	1675	3269	5622	353	3971	1612	2024	1151	1405	1187	1521
Chilpancingo	608	2202	4555	1538	2904	545	1176	676	338	329	454
Chihuahua	1159	659	3012	2784	1361	1294	791	1922	1518	1575	1280
Durango	637	1191	3544	2212	1893	722	571	1350	946	1003	708
Guadalajara	242	1382	3735	1839	174	267	754	977	580	630	342
Guanajuato		1659	4012	1662	2361	145	694	800	357	453	119
Hermosillo	1659		2318	3256	667	1684	1485	2394	1997	2047	1759
La Paz	4012	2318		5609	1616	4037	3838	4747	4350	4400	4112
Mérida	1662	3256	5609		3958	1599	2011	1138	1361	1174	1508
Mexicali	2361	667	1616	3958		2386	2187	2696	2699	2749	2461
Morelia	145	1684	4037	1599	2386		878	737	362	390	157
Monterrey	694	1485	3838	2011	2187	878		1373	925	1021	687
Oaxaca	800	2394	4747	1138	2696	737	1373		499	312	646
Pachuca	357	1997	4350	1361	2699	362	925	499		152	203
Puebla	453	2047	4400	1174	2749	390	1021	312	152		299
Querétaro	119	1759	4112	1508	2461	157	687	646	203	299	
Saltillo	629	1396	3749	2146	2098	809	54	1279	860	937	622
S.L.Potosí	180	1731	4084	1712	2431	360	479	850	411	503	173
Tepic	456	1168	3521	2053	1870	481	968	1191	794	844	556
Tlaxcala	448	2042	4395	1207	2744	385	1016	345	119	-2	294
Toluca	314	1858	4211	1363	2560	201	863	501	126	154	160
Tuxtla Gtz.	1345	2939	5492	794	3641	1282	1626	505	1043	857	1191
Villahermosa	1098	2692	5045	529	3394	1035	1447	574	797	610	944
Xalapa	652	2246	4599	1123	2948	589	907	345	323	168	498
Zacatecas	277	1541	3894	1902	2243	412	434	1040	601	693	398

Figura 4.22 c) Datos de distancias entre estados

CONCLUSIONES

Considero que el objetivo de la tesis se cumplió y supera las expectativas, ya que además de realizar un análisis de este problema se realiza un análisis de las características de los problemas combinatorios y de la teoría de la complejidad computacional que son conceptos que se deben tener muy presentes para entender el problema y además agrego una guía para resolverlo con el objetivo de que puedan resolverlo, personas no expertas, esto a través de un ejemplo práctico, realizando un tour por las 32 ciudades de la República Mexicana, encontrando el recorrido óptimo utilizando Software de apoyo como ARCVIEW3.2 y el WINQSB.

Queda claro que el problema del agente viajero es un problema sumamente estudiado, por dos razones: la primera es por el gran reto que representa encontrar su solución, lo cual implica un gran desarrollo de técnicas que lo resuelvan (en la gran mayoría técnicas heurísticas) y la segunda; es un problema que se toma como base para resolver otra clase de problemas que aunque no son siempre problemas puros del agente viajero, pero pueden atacarse usando variantes de los métodos conocidos para encontrar una solución factible, como lo es el reparto de mercancía en una ciudad. Las situaciones reales pueden complicarse con: más de un vehículo de reparto, horarios de reparto impuestos por los clientes, demoras en los lugares de entrega, capacidades en los vehículos que limitan los recorridos, etc.

Es decir, es un problema que puede transformarse según las restricciones que le agregues o le quites, y se hace en la medida que vas reconociendo el problema y las técnicas de solución que conoces para adaptarlas al problema y así poder resolverlo. Existen variaciones de este problema y de manera general pueden ser: simétrico, asimétrico y puede tener múltiples agentes viajeros o bien uno solo.

Por otro lado es un problema que se le ha explotado en gran medida en un corte netamente teórico dado que es un problema NP-Completo, lo cual significa que es un problema muy difícil de resolver y es por esto que los amantes de la algoritmia y de la curiosidad matemática han visto en el PAV una oportunidad para buscar algoritmos de solución eficientes (técnicas heurísticas) y se ha dejado a un lado el desarrollo práctico. Pero también es cierto que sin investigación el lado práctico o bien de aplicación tampoco se desarrolla. Por ejemplo en otras universidades del mundo como la Universidad de Rice, por mencionar alguna, se forman grupos interdisciplinarios y obtienen investigaciones muy interesantes en donde se observa que hay un vínculo teórico - práctico, como es el caso del Genoma Humano que también ha sido tema de investigación y de aplicación en la Universidad de las Américas campus Puebla (UDLAP).

Y dado que la solución de este problema admite una doble interpretación: mediante grafos y mediante permutaciones, dos herramientas de representación muy habituales en problemas combinatorios, por lo que las ideas y estrategias empleadas son, en gran medida, generalizables a otros problemas.

Cuando resolvemos un problema se trata de encontrar la solución "mejor" u óptima de entre un conjunto de soluciones alternativas, para resolver este tipo de problemas tenemos que elegir entre dos caminos. El primero consiste en buscar la optimalidad con el riesgo de tener grandes, posiblemente impracticables tiempos de computación; el segundo consiste en obtener soluciones con rapidez, aun con el riesgo de caer en la sub-optimalidad. Entre los que siguen la primera opción destacan los métodos exactos y la segunda opción da lugar a los algoritmos de aproximación (heurísticos).

En cuanto las técnicas de solución exactas que existen para resolver problemas de optimización combinatoria se pueden enumerar las siguientes: método simplex, planos de corte, ramificación y acotamiento, enumeración implícita, programación dinámica entre otras. Sin embargo, como se comentaba en el párrafo anterior, su uso es computacionalmente muy costoso.
Ya que el problema es de una naturaleza tal que no se conoce ningún método exacto que resuelva el problema en tiempo polinomial, por lo que ofrecer entonces una solución que sólo sea aceptablemente buena resulta de mayor interés, frente a la alternativa de no tener ninguna solución en absoluto. Es difícil tener un modelo exacto, sin embargo, los algoritmos heurísticos generalmente son más flexibles para resolver problemas donde la función objetivo y las restricciones son más complicadas, permitiendo por ejemplo, la incorporación de condiciones de difícil modelización. Es decir, este tipo de algoritmos resuelve modelos realistas.
De esta manera, no debemos concentrarnos en encontrar su solución óptima, cuando el tamaño de instancia es grande (mayor a 30 nodos), la única opción, es dar una "buena solución" , pero no necesariamente la mejor; por lo que se sugiere elegir una buena técnica heurística.

Dada la investigación que se realizó en cuanto técnicas de solución heurísticas, se encontró que existen una gran cantidad de este tipo de técnicas que se agrupan en: heurísticas de descomposición, inductivos, de reducción, constructivos y métodos de búsqueda local. Y estos últimos junto con los métodos constructivos constituyen la base de los procedimientos metaheurísticos, los cuales hacen una emulación de la realidad, como son el método de Ant Colony system (ANT), en donde el algoritmo se basa en el comportamiento de las hormigas al buscar una ruta más corta entre su nido y la comida. La mayoría de las técnicas metaheurísticas tienen un gran desempeño, pero en los estudios encontrados dentro de las técnicas de redes neuronales, la técnica Red Neuronal de Hopfield, ha resultado ser una de las técnicas menos eficientes en comparación con otras técnicas como Recocido Simulado. Dentro de las técnicas heurísticas una de las primeras en recomendar es la técnica *k-OPT* , en donde *K* representa el número de aristas que se intercambian, y si se incrementa este número en el procedimiento de intercambio el algoritmo se vuelve más exacto, pero el gasto computacional también se incrementa rápidamente con el valor de *K* . Si K es de dos se trata del método $2-OPT$ y tendrá por lo tanto $O\left(n^2\right)$. Además en los estudios comparativos estudiados esta técnica se aleja del óptimo en .1% o .4%, es decir, casi es la solución óptima con la gran ventaja de obtener la respuesta de manera inmediata. Por lo que se sugiere utilizar esta técnica para solucionar el problema del agente viajero, y así se demuestra en algunos de los estudios comparativos realizados.

En la mayoría de los estudios comparativos se centran en dos cosas. La calidad de la solución y el tiempo de solución. Mientras que el desempeño computacional es uno de los criterios más importantes otros criterios deberían considerar una evaluación del software para cualquier problema computacional. Un académico puede estar interesado en un software, con el objetivo de ilustrar varios desarrollos de un algoritmo y practicarlo, resolviendo un problema pequeño sin preocuparse de un código sofisticado. Más aún hacer un mayor uso del Internet y del World Wide Web y de aplicaciones en JAVA, ya que nos ilustran paso a paso el desarrollo del algoritmo, lo cual nos brinda grandes ventajas.

Aún con el uso de códigos, uno podría preferir un lenguaje de programación específico a otro, si hay un lenguaje general propuesto, como FORTRAN, Pascal, C,C++, etc.

O hay programas como AMPL, MAPLE, Matemática, etc. También podríamos usar software con formatos de ingreso y salida de datos muy amigables o con interfaces de gráficas.

Sugerencias

Así, se recomienda realizar un estudio de las ventajas del software que existe para resolver problemas de optimización combinatoria en especial el problema del agente viajero. Que en el caso de este estudio solamente se tomó el software WINQSB y ARCVIEW3.2, en donde se tiene que el primer software utiliza el métodos exacto ramificación y acotamiento y el segundo utiliza el algoritmo Dijkstra mejorado (técnica exacta, que nos ayuda a resolver rutas más cortas entre cualquier par de puntos en una red) apoyada de una técnica heurística de construcción que nos aporta un tour inicial elemental.

La solución que arrojan ambos programas es muy parecida y difieren en 9 Km, lo cual nos indica que la precisión de la información es muy importante y en el caso de ARCVIEW3.2 es un software que cuenta son Sistemas de Información Geográficos que cada vez son más precisos y nos permiten un manejo de información más exacto.

Por otro lado la experiencia que obtengo al cursar la maestría en el posgrado de ingeniería es que la investigación que se realiza en el posgrado es muy baja o casi nula, y considero que hay temas en especial que es necesario estudiarlos y explotarlos dada la gran aplicación que tienen en diferentes áreas, y el problema del agente viajero es uno de ellos, al realizar la investigación previa para armar el estado del arte que había con respecto a éste problema es que no había gran investigación, la última tesis que se publicó en Posgrado de Ingeniería fue realizada en el año de 1985, en la facultad de Ciencias, que es donde se ha realizado más investigación. Se han elaborado cuatro tesis y todas de corte teórico, y es necesario hacer uso de toda la teoría y aterrizarla en alguna aplicación, ya que la investigación y la aplicación van de la mano.

Introducción a la Combinatoria

El análisis combinatorio, pariente pobre de las matemáticas de ayer, no por el valor de los trabajos, sino por el interés que despertaba en el conjunto de los matemáticos, se ha transformado, bajo su forma moderna, en un instrumento esencial. [KAU71]

La Combinatoria si bien es una rama perfectamente definida de las matemáticas, parece haber ocupado un lugar de segunda fila en las preocupaciones de los matemáticos de los siglos pasados.

...Los antiguos griegos la ignoran casi por completo; señalamos sin embargo, que la Geomancia[1] se ocupó de la enumeración y clasificación de configuraciones. Los monjes taoístas que compilaron el libro adivinatorio sagrado, conocido bajo el nombre de Yi-King, tuvieron preocupaciones análogas (en 2200 antes de J.C.); en efecto, el Yi-King describe la tabla del Lo-chou, que no es otra cosa que el cuadrado mágico. (figura A.1):

4	9	2
3	5	7
8	1	6

Figura A.1 Cuadrado mágico

"...Configuración extremadamente notable, si se observa que sumando los elementos de una misma fila, de una misma columna o de una misma diagonal, se obtiene siempre el resultado 15. La fórmula del recuento de combinaciones de n objetos tomados de p en p y sobre todo la fórmula del binomio, que se atribuye a Pascal, habían sido caligrafiadas ya en 1265 por un filósofo persa, Nazir-Ad-Din, como acaba de descubrirse. Y tales ejemplos son innumerables...[2]
Las razones por las cuales los pioneros de la Combinatoria han permanecido oscuros y asilados son numerosas: los descubrimientos (o redescubrimientos) estaban motivados por problemas de naturaleza demasiado diversa; cada vez las preocupaciones y el lenguaje eran demasiado diferentes, las recetas demasiado dispersas.

Cuando LEIBNIZ escribía, a la edad de 20 años, su tratado <Dissertatio de Arte Combinatoria>, buscaba una nueva ciencia con ramificaciones en Metafísica y Moral; pero es bajo el empuje del Cálculo de Probabilidades que PASCAL y FERMAT se preocuparon por el recuento. Son, por otra parte, las preocupaciones topológicas que condujeron a EULER hacia el descubrimiento de las funciones generatrices.

Más cerca de nosotros, la Teoría de los Números había anexionado la Combinatoria: el recuento de las configuraciones se expresa frecuentemente, en efecto, mediante fórmulas aritméticas que poseen propiedades bellas y sorprendentes.
Recientemente, con la aparición de las computadoras y el nacimiento de la Investigación de Operaciones, han surgido aplicaciones completamente diferentes, que llevan a la Combinatoria en una nueva dirección.

---

[1] La Geomancia es una de las artes adivinatorias. Su nombre deriva de dos palabras griegas :ge,tierra y manteia, adivinación; literalmente adivinación a través de la tierra.
[2] KAUFFMAN,A. *Introducción a la Combinatoria. Análisis y aplicaciones.*ª edición,Barcelona,España,1971.

Recuento, Enumeración, Clasificación y Optimización.

La Combinatoria, simplificando al extremo, se puede decir que es el área de las matemáticas donde estudiamos las familias de subconjuntos de un conjunto dado (el que usualmente es finito) que satisfacen ciertas propiedades (*axiomas*). A cada familia de subconjuntos que satisface los axiomas la llamaremos una *solución factible*. [Kaufmann, 1971]

En otras palabras se puede decir que el objeto de la Combinatoria es el estudio de cuestiones del siguiente tipo relativas a conjuntos finitos que poseen una propiedad o una colección de propiedades dadas, es el problema del **recuento**.
Podemos también interesarnos por la lista de los elementos poseyendo esta (o estas) propiedad (s); es el problema de la **enumeración**, y algunas preguntas típicas son: ¿cuántas soluciones factibles diferentes existen? Desde luego, si el recuento da números demasiado elevados (y frecuentemente es el caso de la combinatoria), se renuncia a esta enumeración para realizar solamente una clasificación de los elementos mediante relaciones apropiadas, es el problema de la **clasificación**.

En algunos problemas el conjunto de las soluciones es tal que se le puede aplicar una función de valor y esta función de valor induce entonces un orden total sobre le conjunto; se pueden considerar entonces las nociones de máximo y mínimo y nos encontramos ante un problema de **Optimización** que se plantea de la forma siguiente: cuál es el sub-conjunto de soluciones para el cual la función de valor es máxima (mínima) y cuál es el valor correspondiente.
Para mostrar estos diferentes aspectos en un caso concreto se utilizará un ejemplo donde se darán los resultados provisionalmente y sin demostración alguna.

Se considera un tablero cuadrado (Figura A.3) compuesto por 25 casillas. Se propone colocar 5 fichas en cada casilla de tal manera que haya una y sólo una en cada fila y en cada columna. A cada casilla se le puede asociar una pareja (Xi,Xj) representada por una flecha, así una colocación de fichas sobre el tablero de la figura A.3 podemos asociar un diagrama sagital como el de la figura A.4

Primera pregunta: ¿Cuántas colocaciones diferentes pueden realizarse? No es difícil demostrar que este número es el de las permutaciones de 5 objetos, o sea 5!= 5*4*3*2*1= 120. Se ha realizado el **recuento** de las soluciones. [Kaufmann, 1971]

	A	B	C	D	E
A	5	3	9	2	7
B	3	2	1	4	6
C	1	3	8	2	2
D	3	4	9	6	3
E	9	2	3	1	4

Figura A.5 Asociación de función numérica

	A	B	C	D	E
A		○	○	○	
B	○				
C					○
D					
E					

Figura A.3 Tablero cuadrado

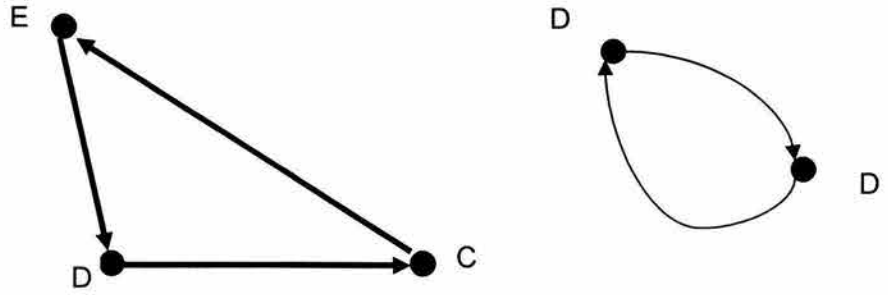


Figura A.4 Diagrama sagital

Se ha realizado el recuento de las soluciones ¿Cuáles son las soluciones, o sea cuál es la lista de estas 120 soluciones?. Una lista de este tipo puede obtenerse fácilmente siguiendo el siguiente procedimiento. Se obtendría:

((A, A), (B, B), (C, C), (D, D), (E, E)),

((A, B), (B, A), (C, C), (D, D), (E, E)),
((A, C), (B, B), (C, A), (D, D), (E, E)),
.....
.....
((A, B), (B, C), (C, D), (D, E), (E, A)).

Esto constituye una **enumeración**.

Supongamos ahora que nos interesamos por la evaluación del número de soluciones (y eventualmente por la lista correspondiente) que tengan estructuras particulares. Examinemos por ejemplo la figura A.4 y llamemos circuito a un camino cerrado de longitud 2 (la longitud es el número de uniones orientadas que forman el circuito). Un problema interesante es el de reunir en clases los circuitos que posean una misma estructura, por ejemplo: 5 circuitos de longitud 1 y 1 circuito de longitud 2 figura A.6. La búsqueda del número de tales clases y su contenido es un problema combinatorio importante. [Kaufmann, 1971]

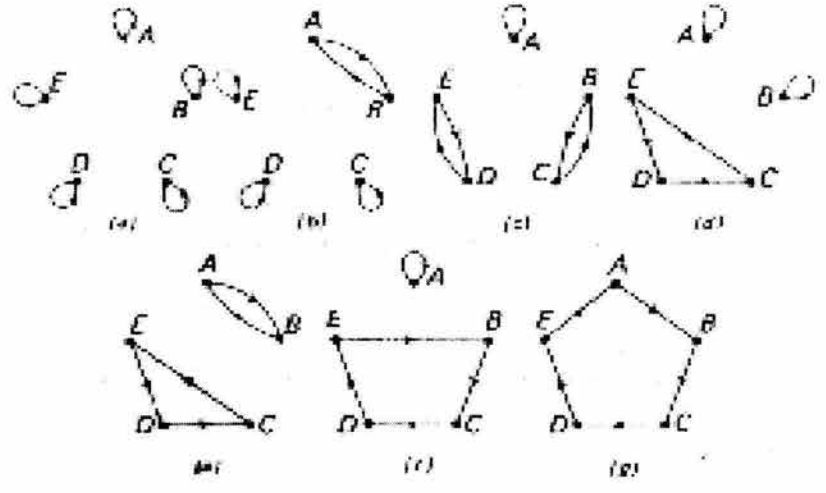


Figura A.61.7 Diagrames sagitales.

Supongamos ahora que se asocie una función numérica a cada una de las casillas (X*i*, X*j*) y que se dé mediante la suma de estos números un valor a toda solución. Busquemos entonces una solución de valor mínimo. Si la función de valor asociada al tablero cuadrado es la de la figura A.5, el valor de una solución indicada en la figura A.3 o en la figura A.5 será:

$$3+3+2+9+1=18$$

En la siguiente figura A.6 se tiene una solución con valor mínimo total igual a 18.

	A	B	C	D	E
A	5	3	9	2	7
B	3	2	1	4	6
C	1	3	8	2	2
D	3	4	9	6	3
E	9	2	3	1	4

	A	B	C	D	E
A	○	○	○	○	○
B					
C					
D					
E					

Figura A.6 Soluciones con valor mínimo

Así pues, se tiene ahora una idea intuitiva del tipo de problemas que se estudian en Combinatoria, como lo es el problema del agente viajero.

Algoritmo Dijkstra

En este anexo se va a explicar el algoritmo de Dijkstra mejorado que es el que ocupa el Software ARCVIEW 3.2 para encontrar las rutas mas cortas entre cualquier par de puntos en una red. Encontrar la ruta más corta entre cualquier par de puntos forma parte de una variedad de problemas prácticos, que pueden ser modelados y resueltos por medio de redes. En estas redes se va a calcular el costo mínimo de enviar flujo de un nodo a otro.

Si $c_{ij} \geq 0$ es el costo unitario del arco A_{ij} que va del nodo al nodo N_i, al nodo N_j entonces, c_{ij} no satisface la propiedad geométrica que dice que el trayecto más corto y por ende más económico entre dos puntos, es el que utiliza la recta que une a esos dos puntos. En términos de la figura se tiene que *no se cumple necesariamente* la siguiente desigualdad

$$c_{ij} + c_{ik} \geq c_{ik}$$

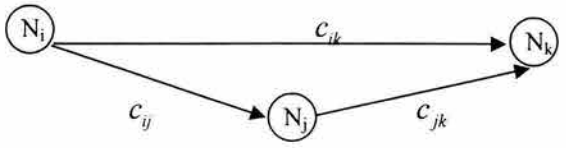


Figura A.1

En una red se puede tener que $c_{ik} \geq c_{ij}$ o bien $c_{ik} \leq c_{ij} + c_{jk}$, dependiendo de los costos unitarios en cuestión. Si el trayecto más económico entre dos puntos es el arco directo que los une, entonces los problemas de redes tendrían soluciones triviales. Afortunadamente, para el desarrollo de esta sección, éste no es el caso en las redes de optimización.
El algoritmo que se va a presentar es el que determina la cadena de arcos más económica de la fuente destino de una red. El algoritmo que se va a aplicar es de Dijkstra. Hay otros métodos que resuelven el mismo problema, tales como los algoritmos de Bellman, Dantzing, Ford y Fulkerson.

Algoritmo de Dijkstra para determinar el trayecto más económico de la fuente al destino de una red

El algoritmo de Dijkstra sirve para determinar la ruta más económica entre la fuente y el destino de una red. Este tipo de problemas, tiene aplicaciones en problemas de distribución y asignación de recursos. Sin embargo, la aplicación más fuerte de los métodos de redes, surge cuando se combinan los problemas de flujo máximo en una red a costo mínimo. Este tipo de problema combinado y sus aplicaciones se verá en la sección siguiente.

En el algoritmo de Dijkstra se considera que los arcos de una red pueden pertenecer a solo uno de los siguientes conjuntos, mutuamente excluyentes, a saber:

a) El arco petenece a un árbol
b) El arco no pertenece a un árbol

Al principio los arcos no pertenecen al árbol. En cada iteración, el algoritmo incrementa en uno el número de arcos en el árbol, hasta llegar a $n-1$ arcos, donde n es el número de nodos en la red. Cuando el árbol queda formado por $n-1$ arcos, el algoritmo llega a su conlcusión y determina la solución del problema.
Estos son los pasos a seguir:

Paso 1.

Sea N_s el nodo fuente. Entonces $L'_{sk} = c_{sk}$ para todo A_{sk} que esté definido en la red. El nodo N_s pasa a ser un elemento del árbol. Se define $L_{ss} \equiv 0$.

Paso 2.

Sea

$$L_{sr} = \underset{k}{Min}\{L'_{sk}\} = \underset{k}{Min}\left[L_{sj} + c_{jk}\right]$$

donde N_k son todos los nodos vecinos[1] a los nodos del árbol en cuestión.

Paso 3.

El arco A_{jr}, pasa a ser un elemento del árbol. Se etiqueta al nodo N_r con (L_{sr}, N_j).

Paso 4.

Si el árbol tiene $n-1$ arcos, pare, la solución óptima ha sido encontrada. En caso contrario continúe con el paso 5.

Paso 5.

Sea

$$L'_{sk} = \underset{k}{Min}\left[L'_{sk}; L_{sr} + c_{rk}\right]$$

para todos los nodos N_k vecinos a los nodos del árbol. Regrese al paso 2.

Este algoritmo también etiqueta a todos los nodos. Un nodo N_j puede tener una etiqueta a todos los nodos. Un nodo puede tener una etiqueta *temporal o permanente*. Independientemente del tipo de etiqueta, cada una de éstas llevará dos componentes. La primera indica el costo temporal o permanente más económico de alcanzar al nodo desde el nodo fuente y la segunda componente indica el nodo del cual procede.
Una etiqueta (L'_{sk}, N_t) es *temporal* mientras que una etiqueta (L'_{sk}, N_t) es *permanente*. Cuando un costo no está definido, se toma a éste como ∞.

Ejemplo. Supóngase que en la siguiente red (después se le da un significado intuitivo) se quiere hallar la ruta más económica del nodo fuente N_s al nodo destino N_t, en donde los

[1] El nodo N_k es vecino del nodo N_t, si es que existe un arco A_{kt} o A_{ik} que conecta a ambos.

números indicados sobre el arco A_{ij} son los costos unitarios c_{ij}. Los arcos sin flecha son adireccionales.

Iteración 1.

Paso 1. N_s pasa a formar parte del árbol, $L_{ss}=0$.

Vecinos a los nodos del árbol son: N_1, N_2, N_3.

$L_{ss}=0, L'_{14}=L'_{41}=2, L'_{s1}=4, L'_{21}=2, L'_{s2}=3,\ L'_{15}=L'_{51}=4, L'_{s3}=1,\ L'_{24}=L'_{42}=5, L'_{32}=1,$
$L'_{4t}=L'_{t4}=4,\ L'_{5t}=L'_{t5}=1, L'_{1s}=\infty,\ L'_{23}=\infty, L'_{3s}=\infty, L'_{st}=\infty, etc.$

Paso 2. $L_{sr}=\underset{k=1,2,3}{Min}\{L'_{sk}\}=\underset{k=1,2,3}{Min}\{L_{ss}+c_{sk}\}=Min\{4,3,1\}=1$

$$L_{sr}=L'_{s3}$$
$$r=3$$

Paso 3. El arco A_{s3} pasa a formar parte del árbol. Se etiqueta a N_3 con $(1,s)$.

Paso 4. Como el árbol no contiene $n-1=7-1=6$ elementos, se continúa en el siguiente paso.

Paso 5. $L'_{sk}=\underset{k}{Min}(L'_{sk}, L_{sr}+c_{rk})$.

Como los nodos vecinos a los nodos del árbol son: N_1, N_2, N_4, N_5 se tiene:

$$L'_{s1}=Min(L'_{s1}, L_{s3}+c_{31})=\min(4,1+\infty)=4.$$
$$L'_{s2}=Min(L'_{s2}, L_{s3}+c_{32})=\min(3,1+1)=2.$$
$$L'_{s4}=Min(L'_{s4}, L_{s3}+c_{34})=\min(\infty,1+\infty)=\infty.$$
$$L'_{s5}=Min(L'_{s5}, L_{s3}+c_{35})=\min(\infty,1+7)=8.$$

Iteración 2.

Paso 2. $L_{sr}=Min(4,2,\infty,8)=2.$

Por lo que $L_{sr}=L'_{s2}$ y $r=2$.

Paso 3. El arco $A_{3,2}$ pasa a formar parte del árbol y la etiqueta de N_2 es $(2,3)$.

Paso 4. Hay dos elementos en el árbol (menor a 6), por lo que se continúa.

Paso 5. Los nodos vecinos del árbol son N_1, N_4, N_5. Entonces

$$L'_{s1}=Min(L'_{s1}, L_{s2}+c_{21})=Min(4,2+2)=4.$$
$$L'_{s4}=Min(L'_{s4}, L_{s2}+c_{24})=Min(\infty,2+5)=7.$$
$$L'_{s5}=Min(L'_{s5}, L_{s2}+c_{25})=Min(8,2+\infty)=8.$$

Iteración 3.

Paso 2. $L_{sr}=Min(4,7,8)=4.$

Por lo que $L_{sr}=L'_{s1}$ y $r=1$.

Paso 3. El arco A_{s1} pasa a formar parte del árbol [2] y el nodo N_1 se etiqueta $(4,s)$.

Paso 4. No se tiene aún la solución óptima.

Paso 5. Vecinos al árbol son los nodos N_4 y N_5 por lo que

$$L'_{s4} = Min(L'_{s4}, L_{s1} + c_{14})$$
$$= Min(7, 4+2) = 6$$
$$L'_{s5} = Min(L'_{s5}, L_{s1} + c_{15})$$
$$= Min(8, 4+4) = 8$$

Iteración 4

Paso 2. $L_{sr} = Min(6,8) = 6$

$L_{sr} = L'_{s4} = 6$

$r = 4$

Paso 3. El arco A_{14} pasa al árbol y N_4 se le etiqueta $(6,1)$.

Paso 4. No se tiene aún la solución óptima.

Paso 5. Nodos vecinos al árbol son N_5 y N_t por lo que

$L'_{s5} = Min(L'_{s5}, L_{s4} + c_{45}) = Min(8, 6+\infty) = 8.$

$L'_{st} = Min(L'_{st}, L_{s4} + c_{4t}) = Min(\infty, 6+4) = 10.$

Iteración 5

Paso 2. $L_{sr} = Min(8,10) = 8$

$L_{sr} = L'_{s5} = 8$

$r = 5$

Paso 3. El arco A_{35} entra a formar parte del árbol [3] y se etiqueta al nodo N_5 con $(8,3)$.

Paso 4. Se continúa.

Paso 5. $L_{st} = Min(L'_{st}, L_{s5} + c_{5t}) = Min(10, 8+1) = 9.$

Iteración 6.

Paso 2. $L_{sr} = L'_{st} = 9.$

$r = t$

Paso 3. A_{5t} entra al árbol y N_t se le pone la etiqueta $(9,5)$.

Paso 4. Como el árbol contiene $n-1 = 6$ elementos, se ha llegado a las solución óptima del problema, que gráficamente aparece a continuación.

[2] Note de la iteración anterior, que, ya sea vía al arco o la cadena. Arbitrariamente se escogió como nuevo elemento del árbol, al arco.

[3] De nuevo existe un empate entre la cadena y la ambas con costo mínimo de 8 unidades. Arbitrariamente se eligió la primera.

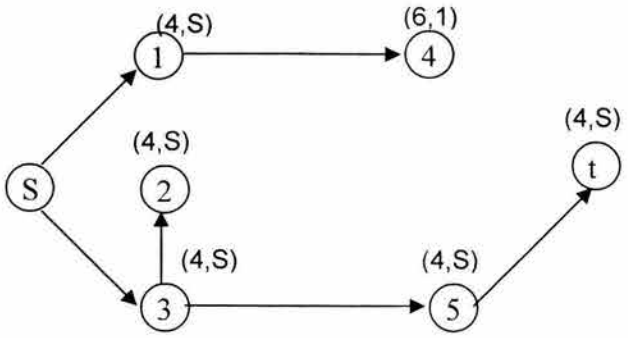


Figura A.2

La solución óptima indica que la ruta más económica del nodo destino a la fuente, es vía la cadena y cuesta 9 unidades por unidad de flujo en esta cadena. Esta cadena óptima no es única. Otra alternativa hubiera sido la. Este problema se podría pensar como una red ferrocarrilera que une varios puntos. Los arcos dirigidos representarían una sola vía, mientras que los arcos no dirigidos pueden representar una doble vía, donde el tráfico de los ferrocarriles se desarrolla simultáneamente en ambos sentidos y los nodos son estaciones.

Algoritmo Dijkstra mejorado

Debido a que el algoritmo sigue siendo el más eficiente para resolver el problema de ruta más corta con arcos no negativos, sería conveniente encontrar una modificación al mismo que permita resolver el problema en redes con arcos negativos de forma eficiente.
En la siguiente sección se explica la modificación al algoritmo Dijkstra para resolver en forma eficiente redes con arcos cuyas longitudes son negativas.

El algoritmo propuesto consta de dos etapas. La primera, llamada de Dijkstra, utiliza el mismo principio que el algoritmo Dijkstra; y la segunda de reescaneo, realiza un reescaneo de los vértices de los subárboles generados durante la primera etapa. Los estados utilizados en este algoritmo son: "no alcanzado" (na), "etiquetado" (e), "permanente" (p), "permanente actualizado" (pa) y "reescaneo" (rs).

Inicio del algoritmo

1. Al principio del algoritmo, el vértice origen s tendrá:
   etiqueta de longitud $d(s)=0$, padre $\pi(s)=s$ y estado $S(s)="etiquetado"$ y el resto de los vértices, $d(v)=\infty$, $\pi(v)=nulo$ y $S(v)="no\ alcanzado"$.

Primera etapa del algoritmo o etapa de Dijkstra

El procedimiento a seguir en esta etapa es muy similar al algoritmo de Diestra original. La diferencia consiste en que para el algoritmo modificado existe un estado "permanente actualizado", que se asigna a nodos con estado "permanente" cuya etiqueta de longitud $d(v)$ no es exacta. Si al término de esta etapa llamada de reescaneo. Los pasos a seguir en esta etapa de Dijkstra son:

2. Se escanea el vértice v con estado "etiquetado" que tenga etiqueta de longitud $d(v)$ mínima. El proceso de escaneo de v consiste en:

2.1 Analizar todos los arcos (v,w) que parten de v, verificando si $d(v)+l(v,w) \prec d(w)$; en caso afirmativo, la etiqueta de w se actualiza $d(w)=d(v)+l(v,w)$ y $\pi(w)=v$.

2.2 Si el estado de w es "permanente" se cambia a "permanente actualizado". Si el estado de w no es "permanente" se cambia a "etiquetado". (En este punto estriba la diferencia de este algoritmo con respecto al de Dijkstra.)

2.3 El estado de v se cambia a "permanente".

3. Si existe algún nodo con estado "etiquetado" se regresa al punto 2. En caso contrario se avanza al punto siguiente.
4. Si existe algún nodo con estado "permanente actualizado" se procede a la segunda etapa (punto 5), en caso contrario el algoritmo ha terminado con la solución correcta.

Observe que los nodos con estado "etiquetado" son los únicos que se escanean, de modo que ningún nodo es escaneado más de una vez en este primera etapa.

Segunda etapa del algoritmo o de reescaneo

Si al final de la primera etapa del algoritmo se tienen nodos con estado "permanente actualizado" quiere decir que no se ha llegado a la solución correcta, debido a que existen nodos con etiqueta de longitud $d(v)$ inexacta y debe procederse a un reescaneo. En esta etapa se reescanean los nodos de los subárboles generados en la etapa anterior. Los pasos a seguir son:

5. Se genera una lista ordenada llamada lista de reescaneo formada por los nodos que pertenecen a los subárboles generados en la etapa anterior. Los subárboles mencionados consisten en los nodos que fueron marcados con estado "permanente" o "permanente actulizado" en la etapa anterior. El orden de la lista consiste en que para un par de vértices v y w, que se encuentren en algún subárbol, en donde v es padre de w, quede antes que w . Si durante el ordenamiento se descubre que un nodo v con estado "permanente actualizado" es descediente de sí mismo, entonces hay un ciclo negativo en la red y el algoritmo se detiene.
6. Se reescanea el nodo que se encuentre al principio de la alista ordenada. El proceso de reescaneo consiste en:

6.1 Analizar todos los arcos (v,w) que parten de v, verificando si $d(v)+l(v,w) \prec d(w)$, en caso afirmativo, la etiqueta de longitud de w se actualiza .

6.2 Si w esta fuera de la lista de reescaneo su estado cambia a "etiquetado" no importando el estado que tenga.

6.3 El estado de v se cambia a "reescaneado" y se saca de la lista.

7. Si existe algún nodo en la lista de reescaneo se regresa al punto 6. En caso contrario se avanza al siguiente punto.
8. Si existe algún nodo con estado de "etiquetado" se procede a la primera etapa (punto 2), en caso contrario el algoritmo ha terminado con la solución correcta.

Dado que el algoritmo de Dijsktra modificado impide el reescaneo de nodos en la etapa de Dijkstra, su tiempo de ejecución crece exponencialmente. En el peor de los casos, el método heurístico de selección de los nodos a escanear, puede no ayudar, de modo que su elección se convertiría en arbitraria. Dado que el algoritmo Bellman-Ford-Moore selecciona los nodos a escanear en forma arbitraria y su límite teórico de tiempo de ejecución es $O(nm)$, el límite de tiempo de ejecución del algoritmo de Dijkstra modificado es $O(nm)$.

Las operaciones del algoritmo se describen a continuación, en la figura A.3

Inicio
/* Se inicializa al nodo origen */
$d(s) = 0$
$\pi(s) = s$
$S(s) = "etiquetado"$
/* Se inicializa el resto de los nodos */
Para todo $(v \neq s)$ **haz** {
$d(v) = \infty$
$\pi(v) = nulo,$
$S(v) = "no\ alcanzado"$
}
/* Primera etapa o etapa de Dijkstra*/

Mientras no se llegue a TERMINA EL ALGORITMO **haz**
Si existe algún con **entonces** {
$u =$ Nodo con etiqueta de longitud $d(v)$ mínima
Para todo arco (u, w) con inicio en u **haz** {
Si $d(u) + l(u, w) \prec d(w),$ **entonces** {
/* Se actualiza w*/
$d(w) = d(u) + l(u, w)$,
$\pi(w) = u$,
Si $S(w) = "permanente"$, **entonces** {
$S(w) = "permanente"\ actualizado"$,
} **sino** {
$S(w) = "etiquetado",$
}
}
}
$S(w) = "permanente"$
}

/* Segunda etapa o de reescaneo */
Si existe algún v con $S(v) = "permanente\ actualizado"$, **entonces** {
Se genera la lista de reescaneo,
Si se encontró un ciclo negativo, **entonces** {

```
      TERMINA EL ALGORITMO,
    } sino {
      Si existe algún nodo v en la lista de reescaneo haz {
        Para todo arco (v,w) con inicio en v haz {
          Si d(v)+l(v,w) ≺ d(w), entonces {
            /* Se actualiza */
            d(w) = d(v)+l(v,w),
            π(w) = v,
            Si w ∉ a la lista de reescaneo, entonces {
              S(w) = "etiquetado",,
            }
          }
        }
        S(v) = "reescaneado",
      }
    }
  } sino {
    TERMINA EL ALGORITMO con la solución correcta,
  }
}  /* Fin del mientras*/
```

Ejemplo del algoritmo

Para entender mejor el procedimiento se presenta un ejemplo. Es importante mencionar que el orden con el cual el algoritmo toma a los arcos de un vértice cualquiera, depende del orden como fueron introducidos los datos.

Se tienen los siguientes datos de la red:

Datos de la red para el ejemplo

Vértice	Arcos	Longitud del arco
1	(1,2) (1,3) (1,4)	-6 -4 -3
2	(2,5)	-6
3	(3,5) (3,2) (3,6) (3,7)	-4 -3 -2 2
4	(4,3) (4,7)	-8 -5
5	(5,9) (5,8)	3 1
6	(6,5) (6,9) (6,10)	1 3 7
7	(7,6) (7,9)	2 -8
8	(8,9) (8,6)	4 2
9	----------	----------
10	(10,9) (10,7)	-6 0

Su gráfica se muestra en la figura A.4

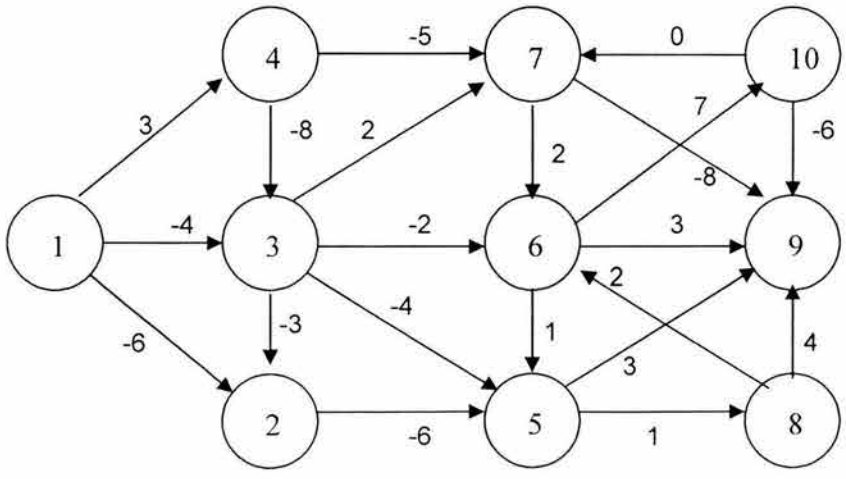
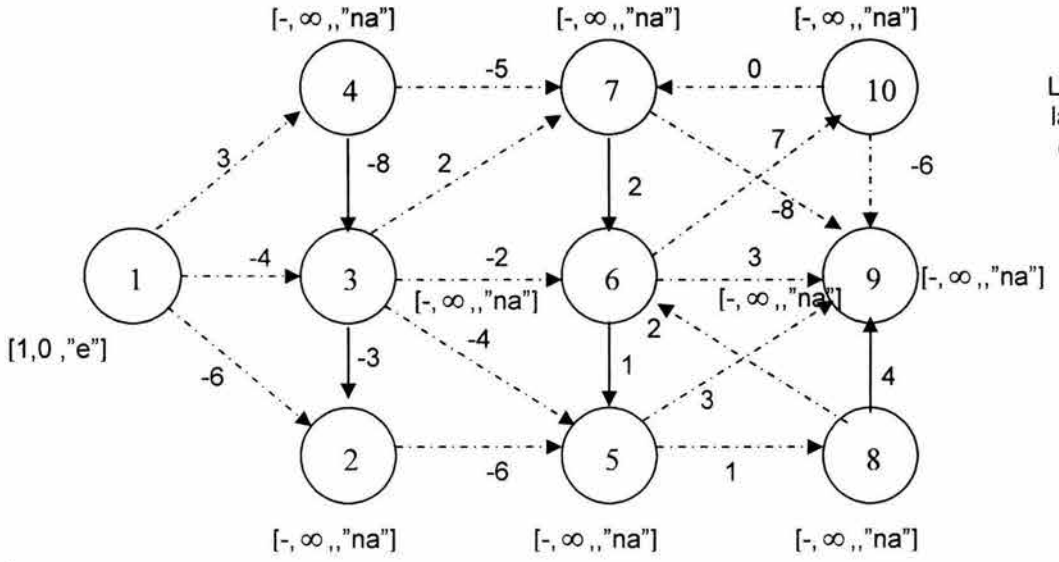


Figura A.4

Al inicio del algoritmo se marca a todos los vértices con padre nulo y con etiqueta de longitud igual a infinito, excepto al vértice origen que en este caso es el vértice 1. Como se observa en la figura A.5

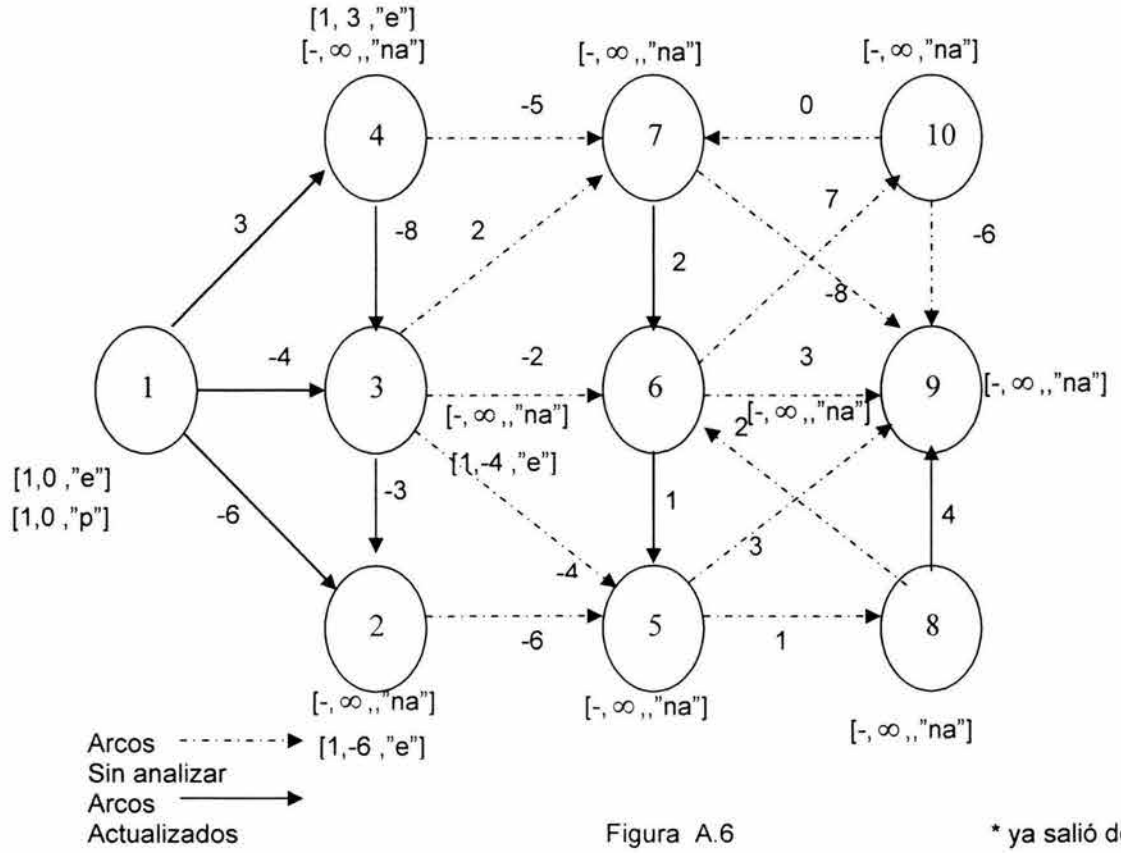


Lista de la 1era. etapa

Nodo
1

Figura A.5

Al inicio del algoritmo el único vértice con estado "etiquetado" es el nodo origen por lo que se procede a escanearlo. Una vez que el nodo se escanea su estado cambia a "permanente". Como se muestra en la figura A.6



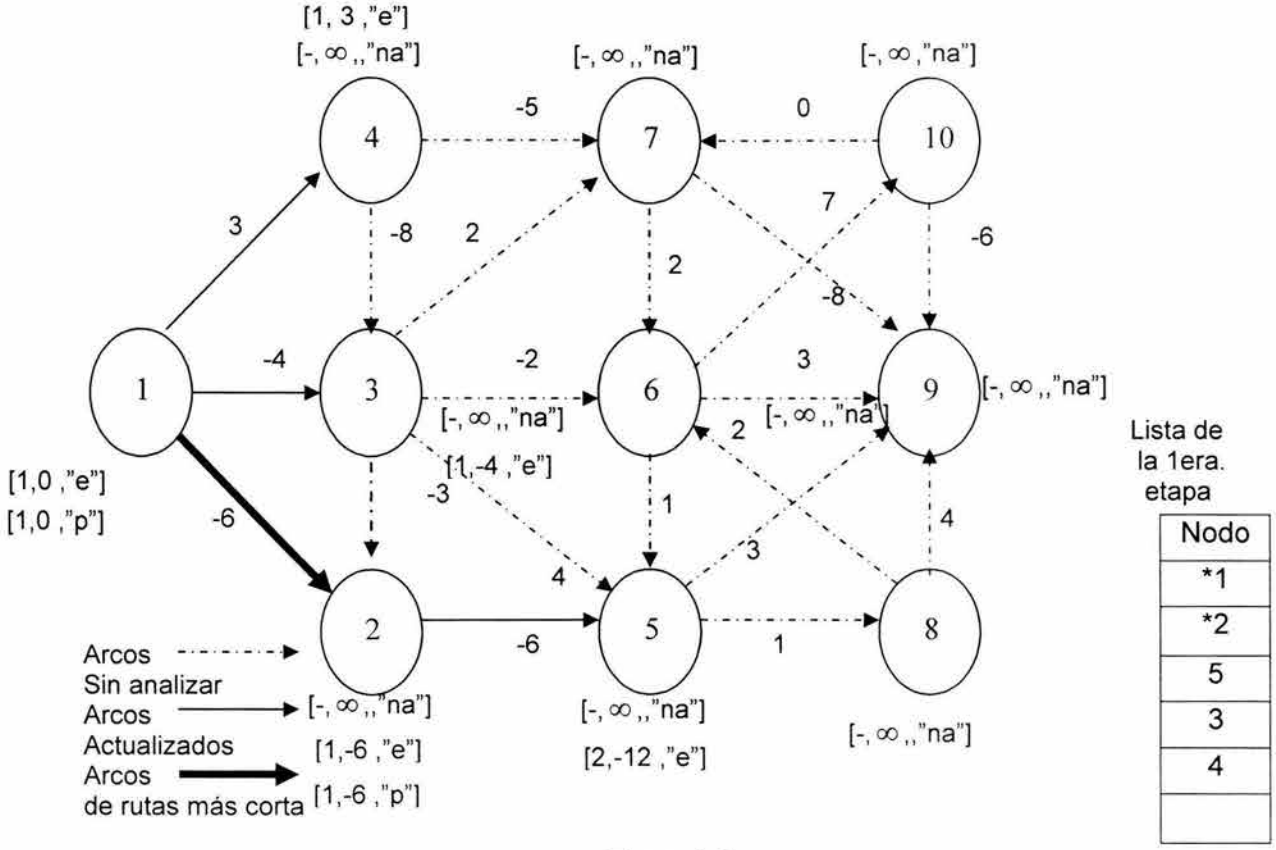
Lista e la 1era. etapa

Nodo
*1
2
3
4

Figura A.6

* ya salió de la lista.

Se escoge el vértice con estado "etiquetado" cuya $d(v)$ sea la mínima para ser el siguiente nodo a escanear. Una vez que el nodo se escanea su estado cambia a "permanente". Como se muestra en la figura A.7.



Lista de la 1era. etapa

Nodo
*1
*2
5
3
4

Figura A.7

* ya salió de la lista.

Si el estado del vértice es "permanente" y llega a él un arco con costo reducido negativo, el estado del vértice cambiará a "permanente actualizado" ("pa") y se *actualiza*.

En nuestro ejemplo, al escanear al nodo 3 con $d(3) = -4$, provoca que la etiqueta de longitude del nodo 2 mejore, pero como su estado es "permanente" se *actualiza* $\pi(2) = 3$ y $d(2) = -7$ y su estado cambia a "permanente actualizado". Ver figura A.8.

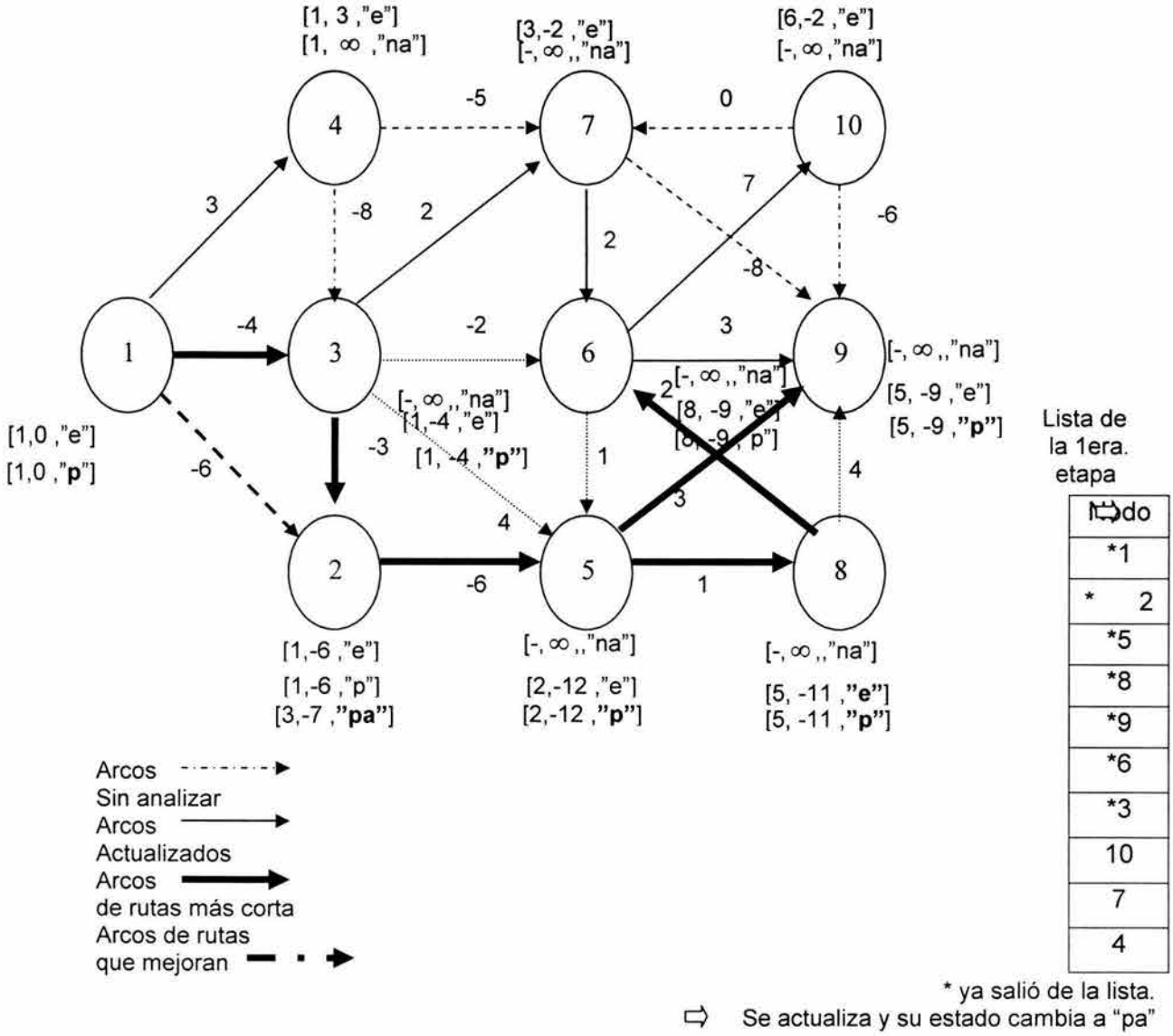


Figura A.8

En la figura A.9 se muestran los diferentes estados que tomaron los nodos y la lista generada hasta llegar al final de la primera etapa.

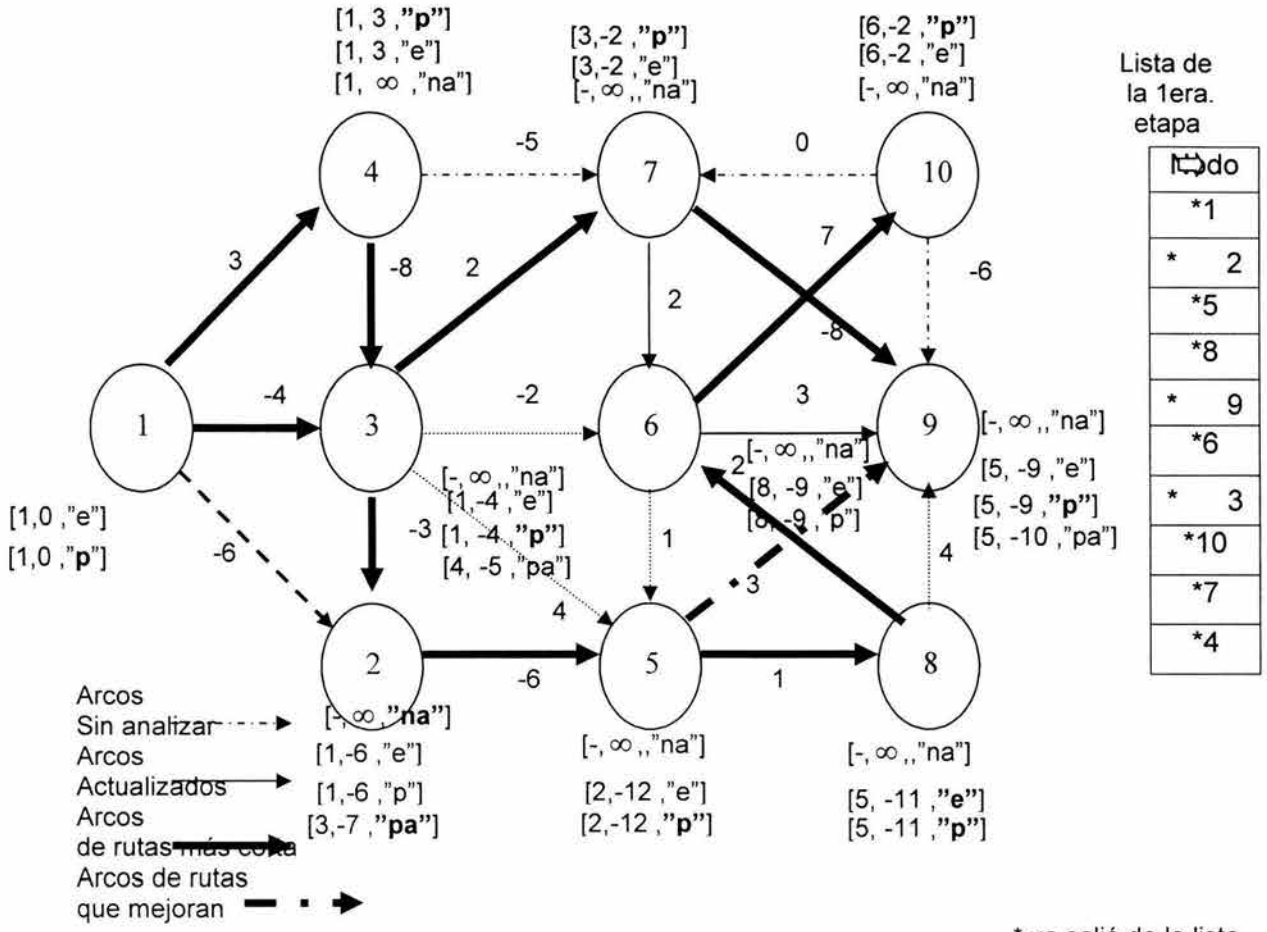


Figura A.9

En la figura A.10 se muestra el subárbol generado en la primera etapa del algoritmo, el cual se forma con los arcos que vienen de los padres que tienen los nodos con estado "permanente" o "permanente actualizado".

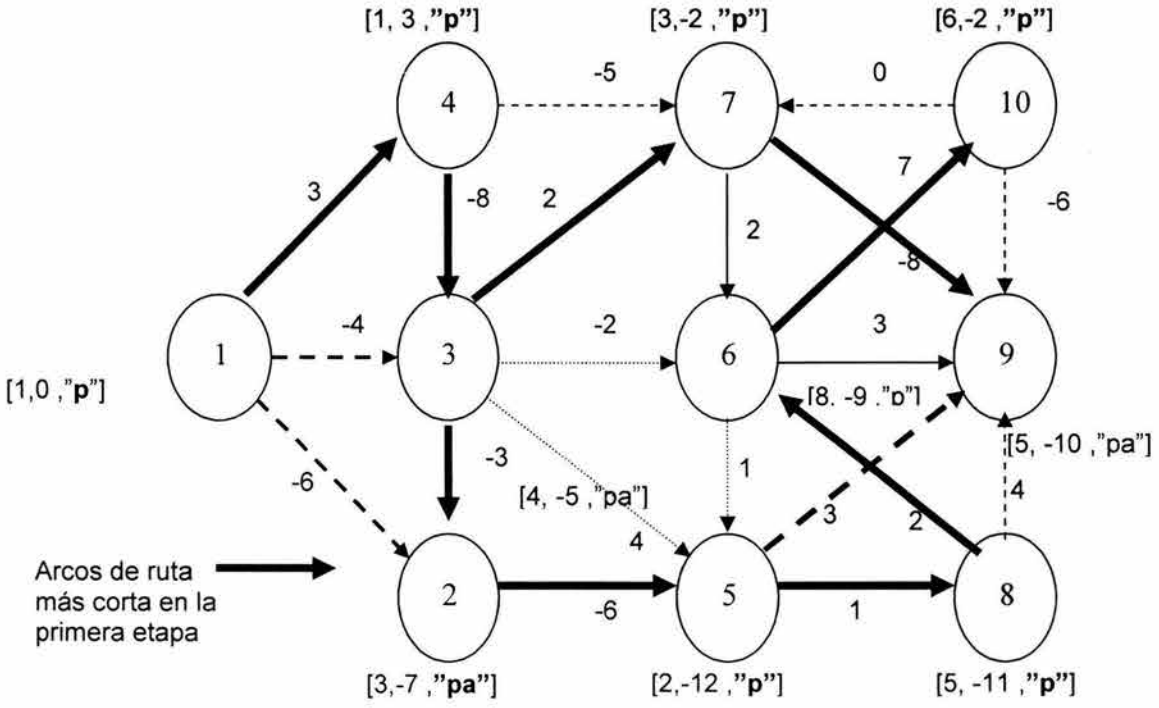


Figura A.10

Si al terminar la primera etapa hay algún nodo con estado "permanente actualizadp" entonces se pasa a la segunda etapa o etapa de reescaneo, donde los nodos serán reescaneados de acuerdo con el orden del subárbol generado en la figura A.10.

Como se puede observar en la figura anterior, se tienen vértices con estado "permanente actualizado", por lo que se pasa a la segunda etapa. En la segunda etapa se ordenan los vértices, de modo que la raíz del subárbol generado, el vértice 1, será el primer vértice a ser escaneado, el vértice 4 al ser el único sucesor del nodo 1 será el siguiente en la lista y así sucesivamente hasta ordenar todos los vértices que se encuentran en el subárbol.
En nuestro ejemplo, para el caso de los vértices 2 y 7; los cuales tienen como padre al nodo 3, el primer nodo que sube a la lista es el último de los hijos del nodo 3; según el orden de entrada de los datos, en este caso es el nodo 7.

En la lista de reescaneo, la cual se muestra en la tabla A.2, los datos para cada nodo se conforman por $v, \pi(v), d(v), S(v)$. En esta tabla se muestra el estado de cada uno de los nodos al iniciar la segunda etapa del algoritmo. Es importante mencionar que el estado que tengan los nodos en la lista de reescaneo siempre será diferente al estado "etiquetado" no importando el estado que tengan.

Lista de reescaneo

Vértice	π	d	S
1	1	0	"permanente"
4	1	3	"permanente"
3	4	-5	"permanente actualizado"
7	3	-2	"permanente"
9	7	-10	"permanente"
2	3	-7	"permanente actualizado"
5	2	-12	"permanente"
8	5	-11	"permanente"
6	8	-9	"permanente"
10	6	-2	"permanente actualizado"

Tabla A.2

Durante la segunda etapa del algoritmo, se escanean y se sacan los elementos de la lista de reescaneo, siguiendo el oden establecido, empezando por el primer nodo de la lista. Al escanear los nodos, su estado cambia a "reescaneado". Los vértices cuyas etiquetas de longitud mejoren y se encuentren fuera de la lista de reescaneo, se marcan como "etiquetados", aunque tengan estado "reescaneado".

En la figura A.11 se muestran los diferentes estados que toman los nodos hasta llegar al final de la segunda etapa. En nuestro ejemplo, al reescanear al nodo 10 se mejora la etiqueta de longitud del nodo 7, cuyo estado era "reescaneado" y se encontraba fuera de la lista de reescaneo, por lo que se *actualiza* y su estado cambia a "*etiquetado*".

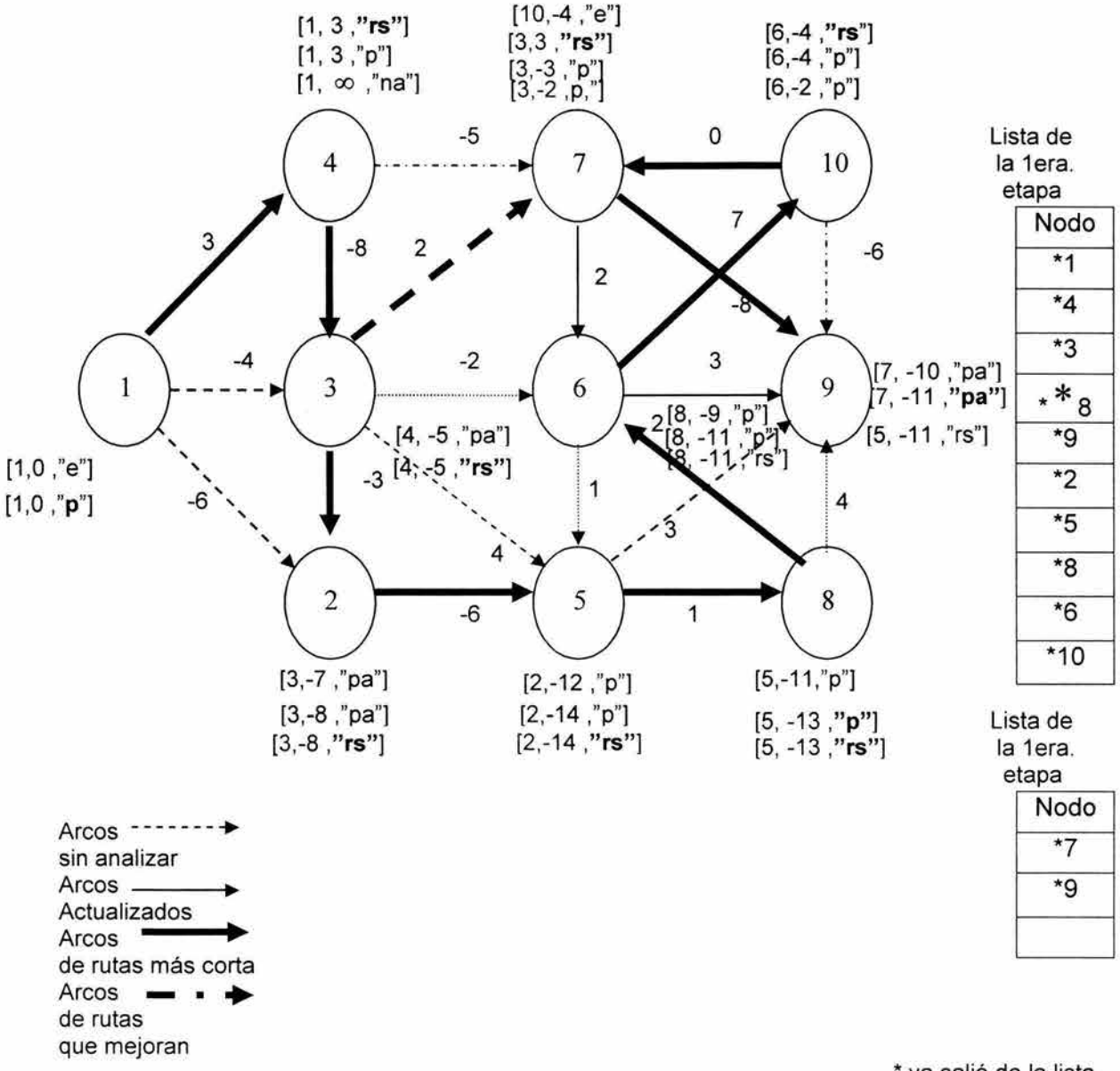


Figura A.11

Dado que existen nodos con estado "etiquetado" al final de la segunda etapa, se continúa con la primera etapa nuevamente. En este caso, el único nodo con estado "etiquetado", al inicio de la etapa de Dijkstra, es el nodo 7, como se muestra en la figura A.12. Es importante mencionar que, en caso de que se tuviera que pasar a la segunda etapa nuevamente, los subárboles a tomarse en cuenta, para la generación de la lista de reescaneo, tendrían su raíz en los vértices con estado "etiquetado" al inicio de la etapa de Dijsktra; como el nodo 7 en este caso.

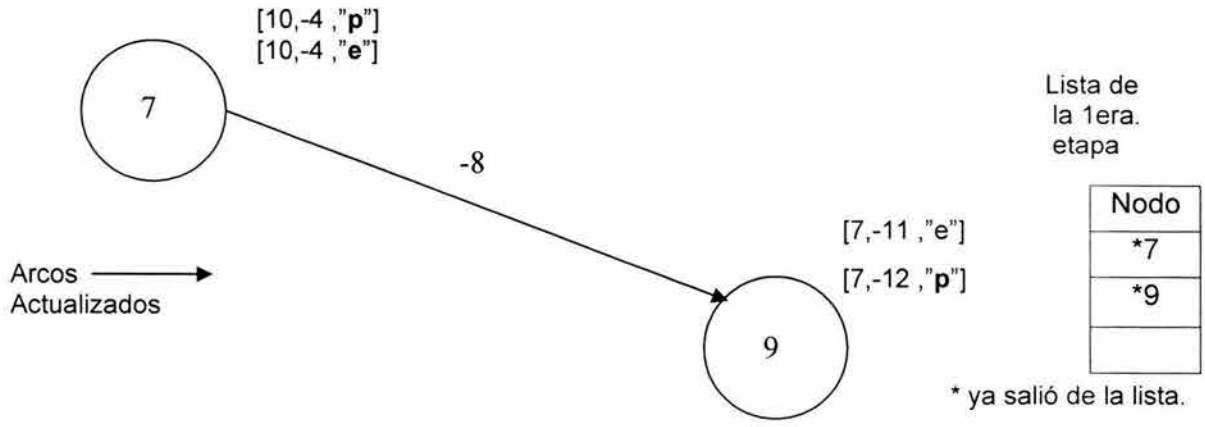


Figura A.12

En la figura A.13 se muestran los estados por los que pasaron los nodos del subárbol generado en esta etapa hasta llegar al fin de la misma.

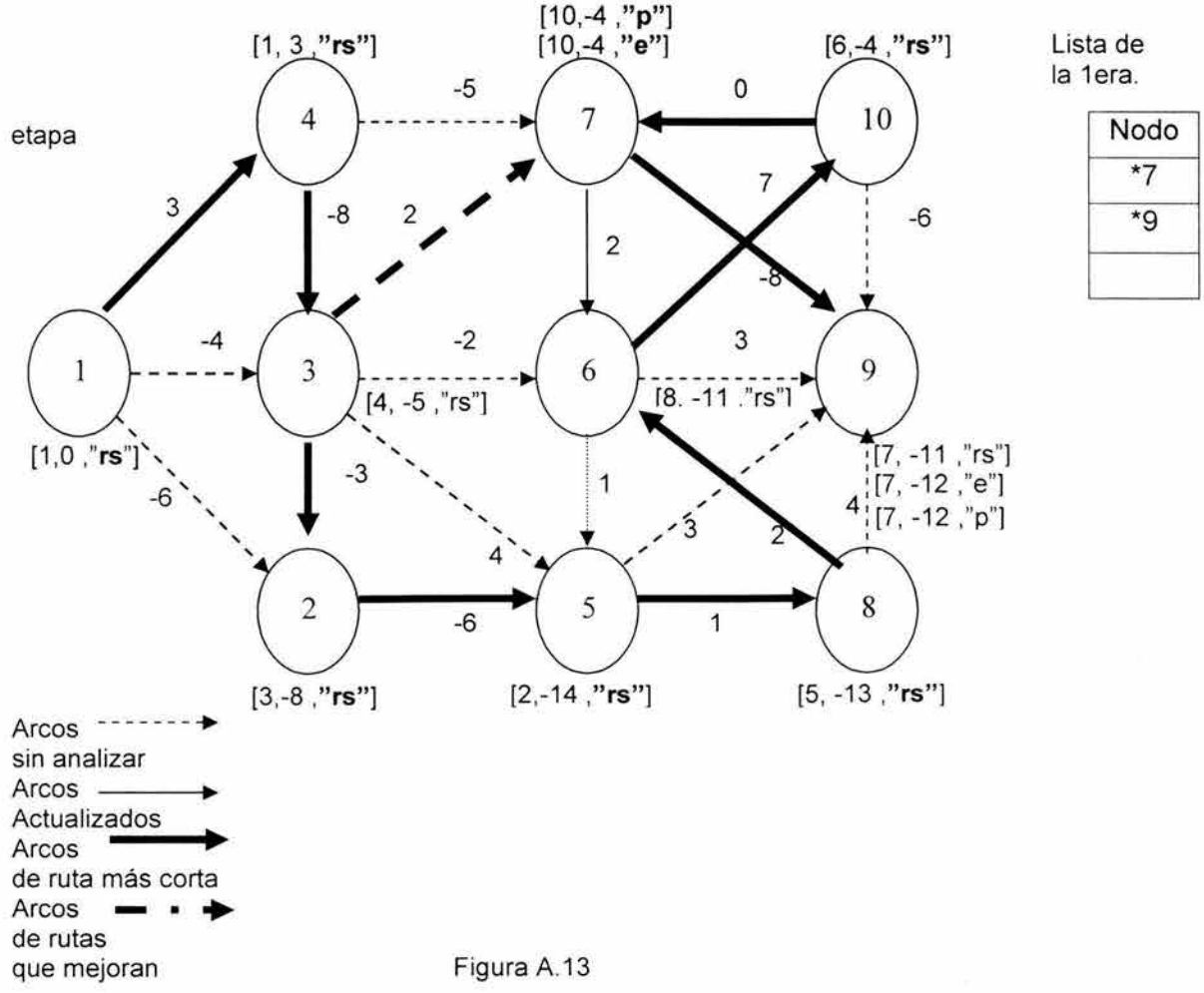


Figura A.13

Al no quedar vértices con estado "permanente actualizado" en la primera etapa del algoritmo, el algoritmo termina.

En la figura A.14 se muestra el resultado final.

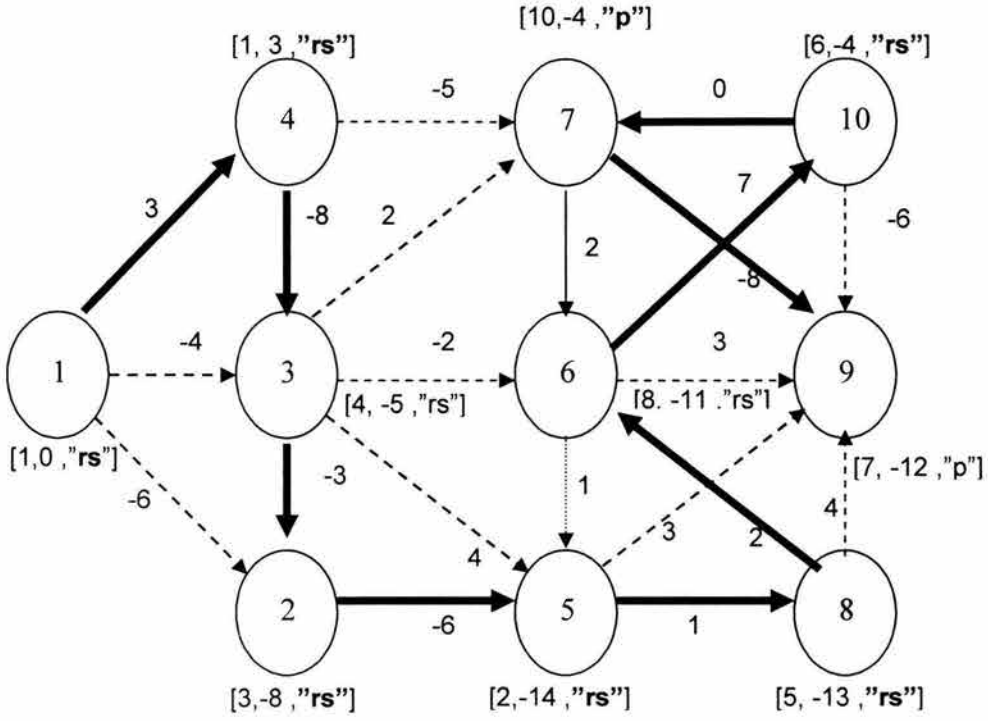


Figura A.14

Bibliografía

[Abellanas, 1991] ABELLANAS, M. *Análisis de Algoritmos y Teoría de Grafos.* Coedición: Macrobitta-ma, México, D.F. 1991.

[Aguirre, 1996] AGUIRRE, Rosalía. *Propiedades y algoritmos para el problema del agente viajero.* Tesis profesional para obtener el título de Licenciada en Actuaría, Universidad Nacional Autónoma de México, 1996.

[Díaz, 1996] DÍAZ, A. (Coordinador). *Optimización heurística y redes neuronales.* Editorial: Paraninfo, Madrid, España. 1996.

[Bazaraa, 1990] BAZARAA, M., JARVIS, J., SHERALI, *Linear Programming and Network Flows*, John Wiley &.Sons, 1990.

[Bellman, 1957] BELLMAN R. *Dynamic Programming.* Princeton University Press, Princeton, New Jersey, 1957.

[Brookshear, 1993] BROOKSHEAR, J.G. *Teoría de la Computación. Lenguajes Formales, autómatas y complejidad.* 1a edición en español, Addison-Wesley iberoamericana, E.U.A., Delaware. 1993.

[Chvatal, 1983] CHVATAL, V., *Linear Programming*, Freeman, 1983.

[Castañeda, 2000] CASTAÑEDA, R.Y. *Estudio Comparativo de diversos métodos de solución del problema del agente viajero PAV.* Tesis profesional para obtener el grado de Maestría en Ciencias con especialidad en Ingeniería en Sistemas Computacionales, Universidad de las Américas, Puebla. Escuela de Ingeniería en Sistemas Computacionales, mayo, 2000.

[Cook, 1998] COOK, W., CUNNINGHAM,W, PULLEYBLANK, SCHRIJVER, A., *Combinatorial Optimization*, John Wiley &.Sons, 1998.

[Feo y Resende, 1995] FEO T. y Y RESENDE M.G.C. Greedy Randomized Adaptive Search Procedures, *Journal of Global Optimization*, 2, 1995, 1-27.

[Festa y Resende, 2001] FESTA, P. Y RESENDE, M.G.C., GRASP: An Annotated Bibliography, *AT&T. Labs Research Tech. Report.* 2001.

[Flores, 2002] FLORES, I. *Apuntes de Programación Entera.* Departamento de Sistemas DEPFI, UNAM, Ciudad Universitaria, México, D.F. 2002. (pag3-7)

[Freeman, 1993] FREEMAN, J. A. *Redes neuronales algoritmos y técnicas de programación*, Ed.:Addison –Wesley, , E.U.A., Delaware. 1993.

[Garey, 1979] GAREY,M. *Computers and Intractability: A guide to the Theory of NP-Completeness.* Freeman, New York, 1979.

[Glover, 1977] GLOVER. F. Heuristics for Integer Programming using surrogate constraints, *Decisión Science*, 1977, 8,156-166.

[Glover, Laguna y Martí, 2000] GLOVER, F., M. LAGUNA Y R. MARTÍ. Fundamentals of Scatter Search and Path Relinking, *Control and Cybernetics*, 29 (3), 2000, 653–684.

[Glover y Laguna , 1997] GLOVER, F. Y LAGUNA, M. , *Tabu Search*, Ed. Kluwer, London.1997.

[Gregory, 2002] GREGORY GUTIN Y ABRAHAM P. PUNNEN. *The Traveling salesman problem and its variations*. Dordrecht : Kluwer Academic, 2002. Combinatorial Optimization, VOL.12

[Gutiérrez, 1991] GUTIÉRREZ, A. *La técnica de recocido simulado y sus aplicaciones*. Tesis de Doctorado. DEPFI, UNAM. Ciudad Universitaria, México ,D.F. 1991.

[Hillier, 2000] HILLIER,F., LIEBERMAN,G., *Introduction to Operations Research*,7ma ed., McGraw-Hill, 2000.

[Jianyu, 1985] JIANYU, Y. *El problema del agente viajero y sus extensiones*. Tesis de maestría, DEPFI, UNAM. Ciudad Universitaria, México, D.F. 1985.

[Johnson y otros, 1989] JOHNSON, D.S., ARAGON, C.R., MCGEOCH, L.A. AND SCHEVON, C. Optimization by Simulated Annealing: An experimental evaluation; Part I, Graph Partitioning, *Operations Research*, 37.1989.

[Jünger, Reinelt , Rinaldi ,1995] JÜNGER, M., REINELT, G. y RINALDI, G. (1995), *The Traveling Salesman Problem, in Handbook in Operations Research and Management Science*, Vol. 7, Ball, M.O., Magnanti, T.L., Monma, C.L. y Nemhauser, G.L. (Eds.), North-Holland, Amsterdam,225–330.

[Kaufmann, 1971] KAUFMANN, A. *Introducción a la Combinatoria y sus aplicaciones*, 1a edición, Barcelona, España, compañía editorial continental, 1971. 617 páginas (pp 349-355)

[Laguna y Martí, 2002] LAGUNA M. Y MARTÍ R. *Scatter Search*, Ed. Kluwer, London.2002.

[Lawler, 1985] LAWLER,E. *The traveling salesman problem: a guide tour of combinatorial optimization*.1a edición, Gran Bretaña, Jhon Wiley, 1985.

[Lokketangen y Glover, 1996] LOKKETANGEN, A. AND GLOVER, F. Probabilistic move selection in tabu search for 0/1 mixed integer programming problems, in *Meta-heuristics:Theory and Practice*, Kluwer, 1996, 467-488.

[Lluch, 1996] LLUCH, J.C. *Introducción a la teoría de grafos y sus algoritmos*. Facultad de Informatica, Universidad Politécnica de Valencia. Valencia, España, 1996.

[Martí, 2001], MARTI,R. *Procedimientos Metaheristicos en Optimización Combinatoria*. Publicaciones del departamento de Estadística e Investigación Operativa. Facultad de Matemáticas. Universidad de Valencia. 2001.

[Medrano, 1973] MEDRANO, L.S. *Teoría de Gráficas*. Programa Nacional de Profesores, Asociación Nacional de Universidades e Institutos de Enseñanza Superior. 1ª reimpresión, México, D.F. 1973.

[Minieka, 1978] MINIEKA, Edwuard. *Optimization Algorithms for networks and Graphs.* 1a edición, Wilbur Meier Dead, School of industrial engineering Purdue University, Indiana, Vol. 1, 1978.

[Nemhauser, 1988] NEMHAUSER,G., WOSLEY,L., *Integer and Combinatorial Optimization*, John Wiley &.Sons, 1988.

[Papadimitriou, 1998] PAPADIMITRIOU,C., STEIGLITZ,K., *Combinatorial Optimization*, Dover, 1998.

[Reeves, 1995] REEVES, C.R. *Modern Heuristics Techniques for Combinatorial Problems*, Ed. McGraw-Hill.UK.

[Osman y Kelly, 1996] OSMAN, I.H. y KELLY, J.P. *Meta-Heuristics: Theory and Applications*,Ed. Kluwer Academic, Boston. 1996.

[Rico, 1999] RICO,J.R. *Esquemas Algorítmicos.* Publicaciones de la Universidad de Alicante, Textos Docentes, España, 1999.

[Rinnoy Kan y Timmer, 1989] RINNOY KAN Y TIMMER. Global Optimization, in *Handbooks in operations research and management science.* Rinnoy Kan y Toods (Eds.), North Holland, 1, 631-632. 1989.

[Rochat y Taillard, 1995] ROCHAT, I. AND E. TAILLARD, Probabilistic diversification and intensification in local search for vehicle routing, *Journal of heuristics*, 1(1), 1995, pp 147-167.

[Schrijver, 1996] SCHRIJVER,A. Theory of Linear and Integer Programming, John Wiley & Sons, 1986.

[Stinson, 19I87] D. R. STINSON. An Introduction to the Design and Analysis of Algorithms;Second Edition (Revised); Winnipeg, Manitoba, Canada. 1987.

[Wolsey, 1998] WOLSEY,L., Integer Programming, John Wiley &.Sons, 1998.

http://www.elbalero.gob.mx/explora/html/atlas/carreteras.html mapa

Referencias en Internet y documentos electrónicos

[www1]. http://www.azc.uam.mx/publicaciones/enlinea2/num1/1-3 .Búsqueda Tabú: Un Procedimiento Heurístico para Solucionar Problemas de Optimización Combinatoria.2001

[www2]. http://www.cs.us.es/delia/sia/htm/98-99/pag-alumnos/webII/indice.html. La investigación operativa. 2000.
[www3]. http://www.azc.uam.mx/publicaciones/enlinea2/num1/1-3

[www4]. http://www.math.princeton.edu/tsp/history/gr120.jpg

[www5]. http://decsai.ugr.es/~castro/CA/node24.html. Complejidad Algorítmica, Juan Luis Castro Peña. Depto. Ciencias de la Computación e Inteligencia Artificial. Universidad de Granada. 1999.

[www5]. http://decsai.ugr.es/~castro/CA/node24.html. Algoritmos y complejidad, Braicovich Gustavo Ariel, Trabajo final. El problema del viajante, 2002.

[www6].http://www.monografias.com/trabajos12/aepqap/aepqap.shtml.Algoritmo Evolutivo Paralelo para Problemas de Asignación Cuadrática – QAP. 2004

[Encarta,2004] Biblioteca de Consulta Microsoft Encarta 2004. 1993-2003 Microsoft Corporation.

Revistas

[González, 1999] GONZÁLEZ V. Y RÍOS M. Investigación de Operaciones en acción: Aplicación del TSP en problemas de manufactura y logística. Ingenierías, mayo-agosto 1999, Vol.II, No.4.

[González, 2000] GONZÁLEZ V. Y ROGER Z. Investigación de Operaciones en acción: Heurísticas para la solución del TSP. Ingenierías, octubre-diciembre, 1999, Vol.III, No.9.