UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

PROGRAMA DE MAESTRIA Y DOCTORADO EN INGENIERIA

FACULTAD DE INGENIERIA

PARTICULARIDADES GEOMÉTRICAS QUE MEJORAN EL DESEMPEÑO DE LOS ALGORITMOS DE BÚSQUEDA LOCAL QUE RESUELVEN EL PROBLEMA DEL AGENTE VIAJERO EUCLIDEANO

T E S I S

QUE PARA OPTAR POR EL GRADO DE:

DOCTORA EN INGENIERIA
INVESTIGACIÓN DE OPERACIONES

P R E S E N T A:

ESTHER SEGURA PÉREZ

Tutor:

Dra. Idalia Flores de la Mota

México, D.F. 2011

Universidad Nacional
Autónoma de México

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Agradecimientos

Quiero agradecer de manera muy especial a mi directora de tesis, la Dra. Idalia Flores de la Mota por el apoyo incondicional, amistad, confianza y sus conocimientos transmitidos para ver materializado este proyecto, además por *sacarme* en más de una.

A mis sinodales por sus valiosos comentarios aportados a este proyecto que sin duda lo mejoraron en gran medida. Y de manera muy especial agradezco al Dr. Javier Ramírez Rodríguez y al Dr. Ricardo Aceves García quienes dedicaron gran parte de su tiempo en la revisión de la tesis y quienes son notoriamente responsables de su versión final.

A la Universidad Nacional Autónoma de México por brindarme el privilegio de formar parte de esta comunidad y porque a través de esta institución he adquirido la profesión y el amor a la docencia.

A CONACyT y al mismo tiempo al proyecto PAPIME PE 102709 por brindarme el apoyo y así culminar mis estudios de doctorado.

A Martha Torres Almazán y Griselda Águilar por llevar mis trámites en tiempo y forma, y sobre todo por su valiosa amistad.

A mis padres, Francisco por ser ejemplo claro de que la dedicación y el trabajo constante conllevan al éxito y a Sara fuente inagotable de amor y tolerancia y quienes juntos me dieron el valor para perseverar y culminar este proyecto.

A Fernando por ser mi compañero y apoyo constante en todo momento, a Sebastián por su amor incondicional y quienes sin tener conocimiento de la Investigación de Operaciones me han llevado siempre a tomar la mejor decisión.

A mis hermanos por estar en el momento justo.

A mis amigos por los grandes momentos compartidos.

A mis compañeros de la CONDUSEF quienes me alentaron a culminar y a Sara Gutiérrez López Portillo por su amistad, confianza y sobre todo por compartirme con su ejemplo que las metas siempre se pueden alcanzar.

Dedico este trabajo:

A mis padres *Sara* y *Francisco*
Por su amor, apoyo, confianza y paciencia infinita

A mis *hermanos*
Por alentarme en todo momento

A *Fernando* y *Sebastián*
Por ser fuente de inspiración

A mis *amigos*
Por los grandes momentos que compartimos

A la *Dra. Idalia Flores de la Mota*
Por el apoyo incondicional, su amistad y los conocimientos aportados a este proyecto

Índice General

	Página
Agradecimientos	i
Dedicatorias	ii
Resumen	iii
Abstract	iv
Capítulo 1: Marco general de la investigación	
1.1 Introducción	2
1.2 Antecedentes	2
1.3 Análisis del estado del arte	4
1.4 La técnica 2-Opt y sus variantes	10
1.5 Planteamiento del problema	16
1.6 Justificación	17
1.7 Metodología propuesta	18
1.8 Objetivo general	18
1.9 Objetivos específicos	19
Capítulo 2: Marco conceptual	
2.1 El problema del agente viajero y su historia	21
2.2 El PAV y la complejidad computacional	34
2.3 Tabla resumen de las instancias resueltas hasta el momento	43
Capítulo 3: El problema del agente viajero euclidiano	
3.1 Definición y formulación matemática del PAVE	46
3.2 Fundamentos de las técnicas de solución heurísticas	47
3.3 Algoritmos de búsqueda local en el problema del agente viajero euclidiano	49
3.4 Algoritmos y esquemas de aproxi*ma*ción-α para el *PAVE*	54
3.5 Esquemas de aproximación	55
3.6 Técnicas Hiperheuristicas	55
3.7 El PAVE y la psicología	56

Capítulo 4: La Geometría Computacional y la solución del PAVE

4.1 Antecedentes 58

4.2 Geometría computacional 60

4.3 Optimización Combinatoria vs. Métodos Geométricos 61

4.4 Paradigmas y estructuras de dato geométricas 62

4.5 Estructuras de datos geométricos 67

4.6 Intersección de segmentos de línea 76

Capítulo 5: Experiencia computacional

5.1 Condiciones de inicio de la metodología 87

5.2 Metodología propuesta 87

5.3 Proceso para la identificación de las aristas que se intersecan 90

5.4 Desarrollo de la metodología propuesta 93

5.5 Resultados obtenidos 96

Conclusiones y futura investigación 115

Bibliografía 119

Anexo I: Laboratorio TSPLAB
Anexo II: Código fuente JAVA

Resumen

En este proyecto de investigación se aborda el problema especial del agente viajero ó The Travelling Salesman Problem como lo conocen a nivel mundial (TSP) en su versión simétrico y euclideano. El cual el problema general puede ser enunciado como: "*Dado un número finito de ciudades, así como el costo asociado de viajar entre cada par de ellas, encontrar el camino con menor costo asociado, de manera tal que se visiten todas las ciudades una sola vez y se regrese a la ciudad de origen*".

En este caso especial el problema del agente viajero euclideano (PAVE) las distancias están representadas por la distancia euclideana entre dos puntos y las ciudades son puntos con ubicación en las coordenadas x e y.

Aún cuando el PAVE resulta muy intuitivo y fácil de comprender, encontrar la solución óptima puede ser una tarea impracticable y en muchos casos imposibles de obtener. Así, el **problema a resolver** en este proyecto de investigación consiste en dar respuesta al problema del agente viajero euclideano, es decir, **encontrar un circuito hamiltoniano mínimo**, en un **tiempo razonablemente bueno**, además de que la **calidad de la solución** quede dentro de los **estándares aceptados**, tomando a las técnicas heurísticas para este cometido.

Es un problema de optimización combinatoria cuya característica es que su espacio de soluciones (vecindad) crece de manera exponencial conforme se agrega ciudades al problema. En la década de los 60´s y 70's se procura encontrar la solución óptima mediante técnicas de programación lineal y en especial de programación entera, sin embargo el tiempo de solución puede ser una tarea infactible. Por lo que a mediados de los 70´s y 80´s se opta por encontrar una solución de buena calidad a través de las técnicas heurísticas, sin garantizar el óptimo global a cambio de respuestas casi inmediatas.

Una de las técnicas que resulta ser de las más eficientes es la técnica 2-Opt, catalogada dentro de las técnicas heurísticas de búsqueda local o llamada también técnica de mejora.

En este sentido, la técnica de solución heurística que se propone es el algoritmo *2-Opt*, y para hacerlo más eficiente se incluyen elementos de la geometría computacional como la identificación de pares de aristas que se intersecan, a través del paradigma de *barrido de plano* y *fuerza bruta* para disminuir la vecindad construyendo una la lista candidata (que es de gran importancia), que contiene pares de aristas que se intersecan.

La metodología propuesta se compara con un conjunto de siete técnicas quienes también han modificado a la técnica 2-Opt., logrando encontrar una metodología hibrida que encuentra una buena solución en un tiempo que resulta ser el mejor dentro de este conjunto de técnicas.

Abstract

This research project addresses the special problem of Travelling Salesman Problem as it is known globally (TSP) in its symmetric version and Euclidean R2. The which the general problem can be stated as: "Given a finite of cities, as well as the cost associated to travel between each pair of them, find their way with lower associated cost, so is visiting all cities only once and return to the city of origin". In this particular case the Euclidean Travelling Salesman problem R2 (ETSP) distances are represented by the euclidean norm between two points and the cities are located in the coordinates x and y.

Even though the ETSP is very intuitive and easy to understand, to find the optimal solution can be a task impractical and in many cases impossible to obtain. The problem to be solved in this research project is to give answer to the problem of the agent traveller Euclidean R2, in other words, find a minimal Hamiltonian circuit, in a time reasonably well, that the quality of the solution to be within accepted standards, taking a heuristic techniques for this purpose.

Combinatorics whose characteristic is that their space of solutions (neighbourhood) grows exponentially as adds cities to the problem is an optimization problem. In the late 70's and the 60´s seeks to find the optimal solution using linear programming and integer programming in particular techniques, however the time of solution can be an infactible task. So in the middle of the 70´s and 80´s you choose to find a solution of good quality through technical heuristics, without guaranteeing the global optimum for almost immediate response. One of the techniques that turns out to be the most efficient is the technical 2-Opt, catalogued in local search techniques . In this sense, the technique of heuristic solution proposed is 2-Opt algorithm, and to make it more efficient included elements of computational geometry as the identification of pairs of edges that intersect through the paradigm of clearance level and brute force to diminish the neighbourhood built a list candidate (which is very important)containing pairs of edges that intersect. The proposed methodology compared to a set of seven techniques who have also changed to the technical 2-Opt., managing to find a methodology hybrid that it is a good solution in a time that turns out to be the best in this set of techniques.

Capítulo 1

Marco general de la investigación

1.1. **Introducción**
1.2. **Antecedentes**
1.3. **Análisis del estado del arte**
1.3.1 Las técnicas de solución exactas
1.3.2 Las técnicas de solución heurísticas
1.3.2.1 Las técnicas de construcción
1.3.2.2 Las técnicas de búsqueda local
1.3.2.3 Algoritmos y esquemas de aproximación
1.3.2.4 Técnicas Metaheuristicas
1.4 La técnica *2-opt* y sus variantes
1.5 Planteamiento del problema
1.6 Justificación
1.7 Metodología propuesta
1.8 Objetivo general
1.9 Objetivos específicos

APLICACIÓN DE LA GEOMETRIA COMPUTACIONAL A LOS ALGORITMOS DE BÚSQUEDA LOCAL PARA MEJORAR LA SOLUCIÓN DEL PROBLEMA DEL AGENTE VIAJERO EUCLIDEANO SIMÉTRICO

Esther Segura Pérez

Secretaría de Postgrado e Investigación, Facultad de Ingeniería, UNAM, 2011.

1.1 Introducción

El análisis del estado del arte, el planteamiento del problema, la metodología propuesta, así como los objetivos de este proyecto de investigación se plantean a lo largo del primer capítulo. La descripción, taxonomía, variantes e historia del problema del agente viajero general se detallan en el capítulo 2 y la descripción del problema del agente viajero euclidiano se describe en el capítulo 3 en donde se destacan las propiedades naturales del problema además que un circuito libre de cruces, es un tour con una mejor calidad que con alguno que si tenga, en el capítulo 4 se destacan los elementos de la geometría computacional como la estructura de casco convexo y el paradigma del barrido de plano que permiten mejoras substanciales tanto en tiempo y calidad de solución, de las técnicas heurísticas. Por último en el capítulo 5 se describe la metodología propuesta y los resultados obtenidos, y se plantean finalmente en otro apartado las conclusiones, alcances y limitaciones.

Además se incluye un anexo, en el que se detalla el ejecutable programado en Visual Basic6.

1.2 Antecedentes

Planear un viaje de una ciudad a otra, o de un país a otro, realizar recorridos visitando pueblos es una de las actividades que se ha venido desempeñando desde siempre, y estas a su vez han sido siempre acompañadas de limitantes de recursos, ya sea de tiempo o dinero.

En el caso del problema del Agente Viajero (PAV) o The Travelling Salesman Problem como lo conocen a nivel mundial (TSP) puede ser enunciado como: "*Dado un número finito de ciudades, asi como el costo asociado de viajar entre cada par de ellas, encontrar el camino con menor costo asociado, de manera tal que se visiten todas las ciudades una sola vez y se regrese a la ciudad de origen*".

En donde **el problema consiste en encontrar el circuito de menor longitud con la finalidad de optimizar recursos**. Es un problema que es fácil de enunciar pero muy difícil de resolver, son dos características que tienen los problemas de optimización combinatoria. En el caso particular de este proyecto, se tratará con el Problema del Agente Viajero Simétrico (PAVES) el cual, tiene como característica que los costos de viajar de la ciudad

C_{ij}, representa el mismo costo que viajar de la ciudad C_{ji}. Este es un caso particular del PAV Asimétrico [1].

Figura 1.1 El problema del Agente Viajero

Alrededor del año de 1850 dos matemáticos británicos William Rowan Hamilton, y Thomas Penyngton Kirkman trataron este problema desde un punto de vista gráfico, originado en un juego inventado por Hamilton. 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 requería que se pasara una sola vez por cada vértice o ciudad, usando solamente las caras del dodecaedro y se regresara al punto inicial.

Sin embargo en 1832 se imprimió un libro en Alemania titulado "Der handlungsreisende,wie er sein soll und was er zu thun hat,um Aufrträge 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 ". "Que debe hacer el agente de viajes para obtener comisiones y ser exitoso en su negocio, escrito por un veterano agente de ventas veterano" [1]. Aunque el libro trata de otros problemas, en el último capítulo hace énfasis en la esencia del problema del agente viajero, encontrar un camino corto dentro de varias opciones y no visitar las ciudades más de una vez.

Y no fue hasta 1920 cuando el matemático y economista Karl Menger lo dejara entrever y lo publicara entre sus colegas en Viena, como el problema del mensajero. En 1930, el problema reapareció en los círculos matemáticos de Princeton y hasta 1931-1932 aparece propiamente como "traveling salesman problem". En 1940, ya había sido estudiado por los estadistas Mahalanobis (1940), Jessen (1942), Gosh (1948), Marks (1948) y el matemático Merill Flood lo popularizó entre sus colegas en la corporación RAND [1]. Eventualmente el PAV iba ganando notoriedad como un problema prototipo de problemas difíciles en optimización combinatoria.

Hoy en día, el PAV, sus variantes y casos especiales como el PAVE, son problemas prominentes en Optimización Combinatoria (OC) y sirven de *inspiración* para implementar nuevas estrategias de solución en el campo de las técnicas heurísticas.

1.3 Análisis del estado del Arte

Debido a la gran complejidad que conlleva resolver este problema, conocido dentro del campo de ciencias de la computación como un problema NP-Completo, Karp [143] y aún el PAVE, siendo un problema NP-Difícil demostrado por Papadimitriou en 1977 [144], se han abordado diversas formas de resolver este problema. Por un lado tenemos las técnicas de **solución exacta** como fuerza bruta o técnicas refinadas de enumeración como las técnicas de **ramificación y acotamiento, planos de corte** o **bien programación dinámica**. Estas técnicas nos ofrecen una solución óptima al problema aunque el tiempo en cómputo no es tan eficiente. Ya que si formamos todas las posibles combinaciones de tours, en este caso $(n\text{-}1)!$, donde $n!=n(n\text{-}1)(n\text{-}2)(n\text{-}3)\ldots(2)(1)$ 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 a grosso modo $f(n)=(n)!$ .Esta función programada en una computadora nos da un costo exponencial, sin embargo, puede sufrir modificaciones de tal manera que se reduzca el tiempo, pero, sigue prevaleciendo la dificultad en la explosión combinatoria. Por ejemplo, si el problema considera la ciudad de la cual partir la función se expresaría como $f(n)= (n\text{-}1)!$, y si se considera el PAVE simétrico entonces se obtiene $f(n)= ((n\text{-}1)/2)!$ Se dice que 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 [17].

Este hecho fundamental se refleja en los tiempos de ejecución de los algoritmos propuestos hasta hoy en día, ya que requieren de un tiempo de computadora exponencial en el número n de ciudades para resolver el PAV de manera óptima.

Para salvar esta dificultad computacional hay dos formas de resolverla:

a) 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.

b) Empleando métodos de solución aproximados pero rápidos (en tiempo polinomial e incluso lineales), que pueden lograr soluciones *subóptimas* que estén aceptablemente cercanas a la óptima.

1.3.1 Técnicas de solución exactas

En cuanto a la primer opción en 1960 Bellman, [145], [68], [146], y [147], utilizan la programación dinámica obteniendo un tiempo $O(n^2 2^n)$. En [148] proponen dos algoritmos que encuentran la solución óptima al PAVE en un tiempo $O(\text{k!kn})$ y $O(2^k k^2 n)$ utilizando esta misma herramienta.

Para el año de 1963 ya habían salido a la luz artículos muy importantes en relación a la técnica ramificación y acotamiento , y la programación lineal, como el elaborado por [71], y [72] en los cuales Little y Murty acuñen el nombre de ***branch-and-bound*** (ramificación y acotamiento). Hubo en especial en el año de 1967 una proliferación importante de documentos relacionados que se ocupan principalmente de resolver al *PAV* de manera exacta a través de la programación lineal. Por ejemplo [79], [80], [81] y [82], [83] y [84], [85], [86] y [87].

En especial el método de ramificación y acotamiento ha resultado ser una de las técnicas exactas de mayor uso para resolver 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 de problema específico que se desea resolver, para así formar un algoritmo de solución adecuado. Existen propuestas de "ramificación y acotamiento" como la de [6] y [7].

Sin embargo, tales técnicas 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, tal como lo muestra [88].

En [95] proponen hibridar técnicas exactas con algunas técnicas heurísticas como *2-Opt*. Los autores mencionan sobre la gran efectividad que tiene la relajación del problema de asignación y proponen la relajación utilizando la técnica *2-Opt* para problemas simétricos y prueban su idea en problemas euclidianos teniendo 20 nodos. En [97] describen un método muy práctico para encontrar soluciones al *PAV* geométrico. Este método involucra la relajación del problema de asignación, inserción de nodos, la técnica *2-Opt* y una rutina de mejora regional (donde las personas identifican una región con sus tours actuales que caen en un subóptimo y entonces se llama a la subrutina para optimizar la región y el tour completo).

Ya en los años de 1992 hasta 1995, se desarrollan trabajos [135], [136]. [137], [138], [139], [140], [141] y [142], destacando la técnica de ramificación y acotamiento como técnica de solución a problemas de gran escala junto con la técnica de planos de corte, utilizando distintos procesadores.

Se utilizan estos algoritmos para resolver instancias de la TSPLIB [WWW1], propuesta por Reinelt en 1991. Este conjunto de 111 instancias son de dominio público y se utilizan para probar algoritmos propuestos y comparar su eficiencia, ya que cuentan con la solución óptima. Este conjunto de instancias está conformado por problemas del agente viajero simétrico y asimétrico, así como por problemas muy relacionados como el problema del circuito hamiltoniano, el problema de ordenamiento secuencial y el problema del ruteo de vehículos. Las instancias simétricas cuentan con distancias euclideanas, y algunas de ellas con distancias especiales del laboratorio de investigación ATT, como las instancias att48 y att532. Las instancias con distancias euclideanas están conformadas por un conjunto de 81 y varían desde 48 hasta 16,000 ciudades.

Del año 2000 a la fecha se han resuelto de manera exacta muchas de las instancias que forman parte de la TSPLIB, utilizando variantes de las técnicas de ramificación y acotamiento y planos de corte en conjunto con técnicas heurísticas, destacándose el grupo de Cook, Chvátal, Padberg y Rinaldi.
La última de las instancias resuelta cuenta con 85,900 ciudades, se resolvió entre febrero de 2005 y abril de 2006 utilizando la técnica de planos de corte para su solución implementados en el software *Concorde TSP Solver* y **136 años de uso** de una computadora para encontrar su solución. En la página siguiente se pueden encontrar todos los detalles de la solución. [WWW2]

Este hecho indica la principal limitante de estas técnicas, ya que, la explosión combinatoria prevalece y estos intentos de generar alternativas relevantes por computadora **no es una tarea factible** además no es suficiente confiar en el poder computacional de alta velocidad de las súper computadoras.

1.3.2 Las técnicas de solución heurísticas

Otra alternativa para salvar la dificultad computacional es proponiendo técnicas de solución heurísticas que encuentran una solución sub-óptima en un tiempo eficiente. Las técnicas heurísticas son procedimientos que nos ayudan a resolver un problema de optimización mediante una aproximación intuitiva, por medio de la naturaleza intrínseca del problema para obtener una buena solución.

Los métodos heurísticos son de naturaleza muy diferentes; por ejemplo, tenemos los métodos de descomposición los cuales descomponen el problema en sub-problemas más sencillos de resolver. Los métodos inductivos pretenden generalizar versiones pequeñas al caso completo. Los métodos de búsqueda local son aquéllos que comienzan con una solución del problema (proporcionada por métodos de construcción o generada de manera aleatoria) y la mejoran progresivamente. Los métodos constructivos son deterministas y consisten en construir paso a paso una solución del problema.

Sin embargo dentro de este conjunto de heurísticas, las que han resultado base fundamental de solución al problema del agente viajero euclideano son, las técnicas de construcción y las de búsqueda local, resaltando también los algoritmos y esquemas de aproximación.

1.3.2.1 Las técnicas de construcción

Las técnicas heurísticas de construcción más reconocidas para resolver el PAVE son: el vecino más cercano propuesta por Rosenkrantz, Stearns y Lewis en 1977, [149] algoritmo de ahorros propuesto por Clark y Wright desarrollados en 1964, y los algoritmos de inserción con sus variantes: el más barato, el más cercano, el más lejano e inserción aleatoria propuestos por Rosenkrantz, Stearns y Lewis (1977), Stewart, 1977, A. W. Boldyreff y Kruskal en [52]. Para dar una idea del comportamiento experimental de estas técnicas, en [24] se muestra un comparativo mostrando que se alejan del óptimo en 18.6%, 9.6%, 16%, 19%,9.9% y 11.1% respectivamente. Resaltando que la técnica más eficiente es la técnica de ahorros y seguida por la técnica de inserción más alejada. Cabe señalar que en el documento no se señala las características del equipo ni el lenguaje de programación utilizado.

1.3.2.2 Las técnicas de búsqueda local

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. De acuerdo con Thomas Stützle [150], Rafael Martí [24], Gregory y Punnen [4] son las que han resultado ser las más exitosas para resolver el PAVE.

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. 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.

Dentro de las técnicas de búsqueda local ampliamente utilizadas y las más exitosas, se encuentran:

- Heurísticas de k-intercambio
  - 2-intercambio o *2-Opt*
  - 3-intercambio o 3-opt
- Vecindades complejas
  - Lin-Kernighan
  - Variante Lin-Kernighan Helsgaun
  - Cadenas de expulsión (ejection chain)

Las heurísticas de búsqueda local se valen básicamente de dos clases principales de mecanismos de mejora del circuito: intercambio de *aristas* e intercambio de *cadenas*. El primero intercambia por ejemplo dos aristas AB y CD por otras aristas AC y BD, cuidando que el circuito resultante siga siendo Hamiltoniano; el segundo mejora el circuito actual moviendo una cadena de tres vértices consecutivos en diferentes lugares (y posiblemente invirtiendo el orden del circuito) hasta que no se puedan obtener mejoras. El proceso se repite con cadenas de dos vértices consecutivos, y después de uno solo. Tanto intercambio de aristas como intercambio de cadenas arrojan resultados razonablemente buenos.

La técnica *2-Opt* es un caso especial del mecanismo de intercambio de aristas *k*-Opt, la cual parte de un circuito hamiltoniano previamente construido por alguna técnica de construcción o generada de manera aleatoria, y para mejorar este circuito, la técnica de búsqueda local toma dos aristas y las intercambia por otras dos generando n^2 rutas posibles, quedándonos con la mejor opción. Así entonces, un *movimiento 2-Opt* consiste en eliminar dos aristas y reconectar los dos caminos resultantes de una manera diferente para obtener un nuevo ciclo. Es Croes en 1958, [55] quien propone por primera vez un método sistemático *2-Opt*; en éste trabajo se citan las bases teóricas de esta técnica de búsqueda local, reportando por primera vez el éxito de los métodos iterativos basados en vecindades aplicados a problemas de optimización combinatoria. En la técnica 3-opt en lugar de tomar dos aristas se toman tres. La generalización de este proceso k-intercambio es debida a Lin y Kernighan propuesta en 1973, [16] especificando la mejor opción de k en cada iteración. 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 $\binom{n}{k}$. Examinar todos los movimientos *k-opt* de una solución lleva un tiempo del orden de $O(n^k)$.

Este conjunto de técnicas explotan las características del PAVE que consisten en intercambiar y reconectar desde un par a un conjunto de aristas, haciendo más eficiente la búsqueda. Se han realizado implementaciones muy sofisticadas que incluyen técnicas de

aceleración de cálculos como las de Johnson y McGeoch propuesta en 1996, [26], Helsgaun en el 2000, [15] y Johnson and McGeoch en el 2002, [12].

A pesar de la superioridad computacional de las implementaciones *k-Opt*, las técnicas de intercambio sencillas como *2-Opt* y *3-Opt* permanecen populares debido a su fácil implementación y excelente desempeño en comparación con el tiempo de cálculo requerido. Estas técnicas son frecuentemente usadas como subrutinas dentro de las heurísticas de mejora que resuelven por ejemplo el problema de ruteo de vehículos (VRP). Estas técnicas constituyen las mejores heurísticas para el PAV euclidiano.

1.3.2.3 Algoritmos y esquemas de aproximación

Se dice que un algoritmo A es un algoritmo de aproximación-α para un problema de optimización Π, con α una constante. Si A es un algoritmo de aproximación tal que para toda instancia I de Π, produce una solución que está dentro de α-veces el OPT (I) [13].

Sahni and Gonzalez en [151] demuestran que el PAVE se puede aproximar al óptimo dentro de un factor constante. El algoritmo con mejor tiempo de aproximación polinomial conocido actualmente sigue siendo el de Christofides propuesto en 1976, [20] que encuentra un circuito de longitud α= 1.5 veces la del circuito óptimo. Se conjetura que la técnica Held-Karp tiene un radio de aproximación de 4/3 Goemans [152], sin embargo la mejor cota es la de 3/2 veces el optimo.
Karp en [153] en un seminario sobre análisis probabilístico de los algoritmos demuestra que cuando n se selecciona de manera uniforme e independiente en un cuadrado unitario, la heurística de disección fija encuentra con alta probabilidad tours cuyo costo están dentro de un factor (1+1/c) del óptimo. Donde c es arbitrariamente largo.

Arora en 1996, [21] diseña un esquema de aproximación para el PAVE y de manera independiente Mitchell [154] en el mismo año encuentra otro esquema de aproximación. El esquema de aproximación de Arora encuentra la solución óptima dentro de un factor (1+1/c) dentro de un tiempo $O(n(\log n)^{(O(\sqrt{d}c))^{d-1}})$.

Los esquemas de aproximación aún cuando ofrecen soluciones cercanas a las óptimas en un tiempo polinomial, son netamente teóricos y cuando se llevan a la práctica no suelen ser tan prometedores. De manera reciente Rodekel y Cifuentes en [155] realizan una implementación al esquema de aproximación de Arora y obtienen resultados no tan alentadores, ya que si bien los tiempos de ejecución son mejores que otros algoritmos la calidad del tour es muy pobre.

1.3.2.4 Técnicas Metaheurísticas

Otra rama importante que resuelve problemas combinatorios son las técnicas metaheurísticas, son estrategias inteligentes para diseñar o mejorar procedimientos heurísticos muy generales con un alto rendimiento. Entre las más populares y bien estudiadas técnicas se encuentran GRASP [30], Recocido Simulado [9] y [31], Búsquedas Tabú [8] y [32], colonia de hormigas ,dónde destaca el trabajo de M. Dorigo [10], Algoritmos Genéticos [33], entre otros [11] y [12]. Las cuales se han aplicado con éxito en la solución de un gran número de problemas reales y en especial a la resolución del PAVE.

El estudio de las técnicas heurísticas demuestra que la búsqueda de la solución óptima forma parte de un deseo que puede ser cubierto únicamente por las técnicas de solución exactas a cambio de esto, las técnicas heurísticas solo pueden otorgarnos una solución sub-óptima en un tiempo razonable.

1.4 La técnica *2-Opt* y sus variantes

Hasta el momento se ha aclarado que hay dos formas básicas de resolver el problema del agente viajero euclideano, mediante técnicas exactas y con técnicas heurísticas. En este proyecto de investigación se utilizaron las técnicas heurísticas debido a que las técnicas exactas acarrean de manera inherente inversión en tiempo que en ocasiones resulta impracticable. Las técnicas heurísticas por otro lado permiten devolver soluciones en la mayoría de las ocasiones muy cercanas al valor óptimo en un tiempo razonablemente bueno.

Dentro del conjunto de técnicas heurísticas que han resultado de mayor relevancia por su éxito prominente son las estrategias de búsqueda local, y dentro de este conjunto se encuentra la técnica *k-Opt* y el caso especial *2-Opt*. En [4] se demuestra que un movimiento *k-Opt* es una serie sucesiva de movimientos de *2-Opt*, en una gráfica completa, cuando $k>2$.

En el año de 1958 aparecen dos artículos que utilizan la técnica de búsqueda local de k-intercambio, Croes [55] que intercambia dos aristas y Bock [192] quien intercambia grupos de tres aristas. En 1965 se publicaron otros dos artículos Reiter y Sherman [77], quienes exploraron diferentes vecindarios, pero fue Lin [76], el primer autor en demostrar el poder del vecindario del algoritmo 2-intercambio o *2-Opt*. Lin encuentra empíricamente que el tour 2-*Opt* de un famoso problema de 48 ciudades, planteado inicialmente por Dantzig [2], puede lograr una respuesta con una desviación del 0.05% con respecto al valor óptimo. Una contribución muy importante en el trabajo de Lin, es que demuestra que el tiempo extra que se requiere para la ejecución cuando $k=4$ no justifica la calidad de la solución, ya que no difiere tanto de la técnica cuando $k=2$, y desde entonces no han aparecido estudios que contradigan esta afirmación.

Existen diversas modificaciones que se han realizado a la técnica 2-Opt para lograr mayor velocidad en el tiempo de corrida del algoritmo y están relacionadas con varios aspectos, la técnica de construcción utilizada, la selección del primer o mejor movimiento de mejora, la estructura de vecindad utilizada, la estrategia de selección de los pares de aristas a intercambiar que modifica el tamaño de vecindad, y los detalles de implementación de la heurística.

En cuanto al aspecto de la técnica de construcción se debe corroborar si es más conveniente proponer un circuito inicial dado por una técnica de construcción del que se sabe que tiene un buen desempeño, o bien generarlo de manera aleatoria. Perttunen en 1994 [162] mostró que el desempeño de las técnicas de intercambio de aristas, en especial la técnica *2-Opt*, muestra mejores resultados cuando se parte de una solución inicial creada por una técnica de construcción con buen desempeño, sin embargo, Babin en 2005 [160] encontró que la técnica *2-Opt* ofrece resultados significativamente mejores que una técnica de intercambio de cadenas, cuando se parte de soluciones generadas de manera aleatoria. Sin embargo, hasta el momento no se ha realizado un estudio de la relación que existe entre la calidad del tour del circuito inicial (construido por la técnica de construcción) y el tour final (el circuito hamiltoniano que se obtiene con la actuación de la técnica de búsqueda local), con el conjunto de instancias de la TSPLIB.

Otro aspecto a tomar en cuenta es de qué forma se selecciona el movimiento de mejora a realizar: puede ser el primer movimiento de mejora detectado en cada iteración o bien el que después de un cálculo de ahorro resulte ser el mejor movimiento de mejora. Estos temas fueron parcialmente contestados por Hansen en 2004 [102] quienes concluyen que para la técnica *2-Opt* realizar el primer movimiento de mejora es ligeramente mejor y más rápida que seleccionar el mejor movimiento si se comienza con un circuito generado de manera arbitraria; se invierten los resultados cuando se parte de un circuito generado con una técnica de construcción.

En cuanto la estructura de vecindad utilizada se basa en movimientos de inserción o intercambio de nodos principalmente. En el movimiento de inserción de nodos se selecciona un nodo v_i y se inserta entre dos nodos adyacentes v_p y v_q, agregando las siguientes aristas (v_p,v_i), (v_i,v_q) (v_{i-},v_{i+}) y borrando las aristas (v_p,v_q), (v_{i-},v_i) (v_i,v_{i+}). En el movimiento de intercambio de nodos los vértices v_i y v_j intercambian de posición agregando las aristas (v_{i-},v_j), (v_j,v_{i+}) (v_{j-},v_i), (v_i,v_{j+}) y borrando las aristas (v_{i-},v_i), (v_i,v_{i+}) (v_{j-},v_j), (v_j,v_{j+}). Ocurre una excepción si (v_i,v_j) es una arista del tour, en el cual el movimiento es equivalente a insertar v_i entre v_j y v_{j+} o insertar v_j entre v_{i-} y v_i. Una generalización de estos movimientos los realizó Bentley [191] y Or [181]. Estos algoritmos garantizan movimientos que preservan la orientación del tour. Para el caso del algoritmo *2-Opt* la estructura de vecindad está basada en intercambiar únicamente dos pares de aristas no adyacentes y reemplazadas por otro par de aristas, y para mantener la orientación consistente del tour, se hace necesario que una de las dos sub-rutas cambie de orientación.

Por ejemplo, al revertir la sub-ruta (v_i, v_{i+} ,,,v_j, v_{j+}) es reemplazada por (v_i, v_j ,,,v_{i+}, v_{j+}). Finalmente el costo de la solución cambia por un movimiento de *2-Intercambio* que puede ser expresado como $\Delta_{ij}=c(v_i, v_j)+(v_i+, v_{j+})-c(v_i, v_{i+})-c(v_j,v_{j+})$. Una solución 2-Opt se obtiene aplicando iterativamente movimientos de *2-Intercambio* hasta que ya no se posible obtener un valor Δ negativo.

En cuanto las estrategias de selección de los pares de aristas a intercambiar que modifica el tamaño de vecindad, Johnson y McGeoch en 2002[12] sintetizan las estrategias fundamentales, que están relacionadas con la generación de una lista candidata que es de vital importancia, ya que estás estrategias permiten modificar la vecindad del espacio de búsqueda.

- Generar una lista restringida de vecinos: únicamente se consideran los vecinos más cercanos de un vértice *p* para llevar a cabo las reconexiones (esta regla fue inicialmente propuesta por Zweig en 1995, [163]. Asumiendo que las ciudades más distantes son poco probables de dar una mejora.

Los criterios de construir esta lista ha variado, algunas de estas ocupan algunos elementos de la Geometría Computacional (GC) , en 1992 Bentley [157] comenzó a insertarlos, así como [171], [174],[176], [177], [178], [179], [183] y [185] los cuales muestran de manera experimental que algunas estructuras de datos geométricos como *fixed radius near neighbor* permiten disminuir el espacio de búsqueda, el cual consiste en fijar una arista de menor costo y con base en esta se eligen las aristas más cercanas a esta, y Reinelt [193] propone la construcción de una *gráfica Delaunay*, la cual provee un conjunto de aristas para inicializar la lista candidata. Enseguida esta lista se expande agregando aristas entre la lista candidata. Otra técnica de construcción de la lista candidata conocida como *gráfica vecina k-cuadrante*, propuesta de manera inicial por Miller y Pekny [194] para el problema de 2-acoplamiento (el cual es una relajación al PAV), y lo utiliza por primea vez Johnson y McGeogh [189] en el contexto del PAVE. En esta gráfica cada vértice v_j es el origen de cada cuadrante en el plano euclideano y k/4 define los vértices más cercanos al vértice v_j. Sea q_{ij} el número de vértices en el cuadrante i para el vértice v_j. Si $\sum_{i=1}^{4} q_{ij} < k$ entonces se llena la lista candidata para v_j con las $k - \sum_{i=1}^{4} q_{ij}$ ciudades más cercanas a v_j no incluidas aun en la lista. Esta lista de candidatos más cercanos es utilizada en las implementaciones más avanzadas de los algoritmos de búsqueda local enviados al 8th DIMACS TSP Implementation Challenge (Concurso organizado para exponer los mejores algoritmos, para resolver el PAV y sus variantes), organizado por David Johnson en el año 2009.

Dentro de este conjunto de estrategias de selección de pares de aristas para disminuir el espacio de búsqueda ninguna técnica a utilizado hasta el momento el elemento de geometría computacional *intersección* de pares de aristas como criterio de construcción de

la lista candidata y así hacer más eficiente en tiempo al algoritmo *2-Opt*. Es por esto que se proponen en este proyecto se utilizan estos elementos.

En este sentido, si centramos nuestra atención al PAVE en instancias de 2-dimensión, y consideramos movimientos *2-Opt* en los pares de aristas que se intersecan en un punto en común, por la desigualdad del triángulo tales movimientos no causan el incremento en la longitud del tour (bajo la norma euclideana, la longitud decrece), la consideración de estos movimientos puede provocar la inclusión de cruces que no se encontraban en el tour original. En Leeuwen y Scoone en 1980, [164] demuestran que en el peor de los casos se puede eliminar todos los pares de aristas que se intersecan en un tiempo $O(n^3)$.

En el caso especial del PAVE por el hecho de que de manera "*natural*" cumple con la desigualdad del triángulo, lo cual quiere decir que siempre podremos acortar los tours dirigiéndonos de manera directa a una ciudad sin pasar por ciudades intermedias. Y si nos basamos en la siguiente observación realizada por Flood en 1956 [156] la cual nos dice que 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. Lo cual implica que siempre se puede mejorar un tour utilizando la propiedad "*natural*" del PAVE.

Otra cuestión importante que concierne a los algoritmos de búsqueda local y de manera especial a la técnica *2-Opt*, es ¿cuántos movimientos se tienen que efectuar antes de alcanzar el óptimo local? En este sentido el algoritmo *2-Opt* puede lograr resultados sorprendentes en algunas instancias, tanto en tiempo como en calidad del tour en un número subcuadrático de pasos [26]. Sin embargo, los análisis teóricos siguen siendo limitados, incluso el análisis del peor de los casos en tiempo sobre instancias euclidianas generadas de manera aleatoria, fue divulgado recientemente por Englert en 2008. [158]

En cuanto los detalles de implementación de la heurística existen estudios experimentales [157] y [160] que muestran cómo se pueden lograr mejoras importantes en tiempo de respuesta de la técnica *2-Opt*. También se han desarrollado trabajos en los cuales se muestran implementaciones de hardware muy eficientes como la propuesta por Mavroidis en 2007 [161]. En especial, en Johnson y McGeoch en 2002[12], se mencionan dos características clave para lograr mayor velocidad en la técnica 2-Opt y en general para las técnicas de *k-Intercambio*:

- Evitar las redundancias del espacio de búsqueda: cuando se implementa la técnica *k-Opt*, se puede ahorrar tiempo incluyendo un cálculo que detecte los intercambios que no mejoran la solución [4].
- Representación del circuito a través de árboles: utilizar este tipo de representaciones del circuito acelera los tiempos de cómputo. Esta representación es muy útil en los

algoritmos de búsqueda local por vecindades ya que se puede identificar de manera más rápida los movimientos que son altamente prometedores. [178] Y se observa en lo siguiente: un movimiento (Intercambio) *2-opt*, básicamente involucra cortar el tour en dos lugares (posiciones) y revertir el orden de uno de los dos segmentos resultantes antes de lograr conectar el tour final. Si el tour se almacena en una forma directa utilizando un arreglo o un lista doblemente ligada, esto significa que el tiempo de desempeño del movimiento *2-Opt*, debe ser al menos proporcional a la longitud del segmento más corto. Empíricamente Bentley [189], encuentra que el segmento más corto crece a razón de $n^{0.7}$ al igual que el tiempo de desempeño de cada movimiento. Si se considera la alternativa de la representación del tour mediante árbol, esto se puede reducir a $\sqrt{n}$ usando árboles de segundo nivel o *logn* usando la estructura de dato de árbol. Esto demuestra la conveniencia de representar al tour como un árbol.

- Utilizar la estrategia de "don't look bits"; se basa en la observación de que si el vértice base v_i y el recorrido vecino por este vértice no cambia después de cierto tiempo, es poco probable que la selección de este vértice produzca un movimiento de mejora. Así, mediante la asociación de una variable binaria (o bandera) con cada vértice, la vecindad está restringida a los movimientos para los cuales los vértices base v_1 se encuentren apagados. El bit del vértice v_i es encendido por primera vez si no provoca ningún movimiento de mejora. Por el contrario está apagado cuando uno de sus vértices adyacentes es utilizado para un movimiento.

El análisis de la literatura muestra algunas propuestas de variaciones al algoritmo *2-Opt*, en especial con la construcción de la lista candidata. Bentley [157], [176],[177],[185] y [191] Johnson y McGeoch [12] ,[189] y [4], realizan las principales propuestas de modificaciones a la lista candidata, utilizando elementos de la geometria computacional, a continuación se resumen las técnicas así como sus características.

En la tabla 1.1 se observa que las modificaciones al algoritmo *2-Opt*, todas utilizan una técnica de construcción para generar el primer tour, en cuanto si se utiliza o no la estrategia de lista restringida, todas las implementaciones excepto la propuesta por Bentley, Concorde y Croes, no lo hacen, a cambio de esto emplean una estructura de datos basada en árbol y hacen la búsqueda de manera directa, sin recurrir a la búsqueda de movimientos prometedores. Dos implementaciones de Jhonson y McGeoch [2opt-JM-20-quadrant-neighbors y 2opt-JM-40-quadrant-neighbors], la propuesta por Croes y la de este proyecto, no utilizamos la estrategia don´t look bits, también se observa que todas las implementaciones utilizan la representación del tour mediante un árbol excepto la propuesta de Croes. Además todas las implementaciones excepto la propuesta por Concorde, utilizan el mejor movimiento de mejora, es decir, después de ciertos movimientos se toma el que resulte con el mayor ahorro. En cuanto la construcción de la lista de vecinos restringida, se cuenta con dos elementos principalmente, partición del

espacio de búsqueda con la estructura de geometría computacional *quadtrees* y la estrategia de *Fixed radius nearneighbor*, sin embargo ninguna de las estrategias implementadas hasta el momento, utilizan el elemento *intersección*, lo cual, puede interpretarse como un área de oportunidad para construir la lista de vecinos restringidos mediante esta estructura.

Caracteristicas	2opt-JM-10-quadrant-neighbors wiht don't-look bits	2opt-JM-20-quadrant-neighbors with don't-look bits	2opt-JM-20-quadrant-neighbors	2opt-JM-40-quadrant-neighbors	2opt-JM-40-quadrant-neighbors	2-Opt:Bentley Algorithm	Concorde-2opt Algorithm	2-Opt Esther	2-Opt Croes
Detalles de implementación									
Heuristica de construcción	✔	✔	✔	✔	✔	✔	✔	✔	✔
Lista restringida de vecinos	✔	✔	✔	✔	✔	✖	✖	✔	✖
Estrategia Don´t look bits	✔	✔	✖	✖	✔	✔	✔	✖	✖
Representación del tour basado en árbol	✔	✔	✔	✔	✔	✔	✔	✔	✖
Movimiento de selección									
Primer Movimiento	✖	✖	✖	✖	✖	✖	✔	✖	✖
Mejor Movimiento	✔	✔	✔	✔	✔	✔	✖	✔	✔
Estructura Geometria computacional (relacionada con la lista restringida de vecinos)									
Fixed radius nearneighbor	✖	✖	✖	✖	✖	✔	✔	✖	✖
Quadtrees	✔	✔	✔	✔	✔	✖	✖	✖	✖
Intersección	✖	✖	✖	✖	✖	✖	✖	✔	✖

Tabla 1.1 Caracteristicas de las principales implementaciones del algoritmo 2-Opt

1.5 Planteamiento del problema

Aún cuando el PAVE resulta muy intuitivo y fácil de comprender, encontrar la solución óptima puede ser una tarea impracticable y en muchos casos imposibles de obtener. Así, el **problema a resolver** en este proyecto de investigación consiste en dar respuesta al problema del agente viajero euclideano, es decir, **encontrar un circuito hamiltoniano mínimo**, en un **tiempo razonablemente bueno**, además de que la **calidad de la solución** quede dentro de los **estándares aceptados**, tomando a las técnicas heurísticas para este cometido. Johnson y McGeoch [189] consideran que una buena heurística es aquella que arroja resultados dentro de un 10% con respecto al valor óptimo.

En este sentido, la técnica de solución heurística que se propone es el algoritmo *2-Opt*, y para hacerlo más eficiente se incluyen elementos de la geometría computacional como la identificación de pares de aristas que se intersecan, a través del paradigma de *barrido de plano* y *fuerza bruta* para disminuir la vecindad construyendo una la lista candidata (que es de gran importancia), que contiene pares de aristas que se intersecan.

Aunado a esta propuesta, se detecta a través del análisis de la literatura tres aspectos que contribuyen al desarrollo de este proyecto de investigación: que la calidad del tour final (la que se obtiene con la ejecución del algoritmo de búsqueda local) depende fuertemente del circuito hamiltoniano inicial, (el que se obtiene con la técnica de construcción) cuando se trabaja con el conjunto de instancias de la TSPLIB [158]. En este sentido, si se parte de un buen circuito inicial, con mayor probabilidad se obtendrá un circuito final de mejor calidad. La estructura geométrica de *casco convexo* permite dar una excelente solución inicial a las técnicas de inserción. Wiorkowski y McElvain en 1975 [180] , Or en 1976[181], Stewart en 1977[182], Norback y Love en 1977[183] afirman que otorgando un sobtour inicial alas técnicas de inserción se tiene como resultado un tour final de mejor calidad. Debido a este hecho, se incluye también el elemento de Goemetría Computacional *casco convexo* en las técnicas de construcción por inserción para obtener un circuito de mejor calidad.

El segundo aspecto es que, las técnicas de construcción como el vecino más cercano [149], algoritmo de ahorros y los algoritmos de inserción con sus variantes: el más barato, el más cercano, el más lejano e inserción aleatoria [52], que han resultado ser de las más utilizadas por el éxito en su desempeño, tienen la gran deficiencia que en los tours finales contienen aristas que se intersecan, lo cual enfatiza la necesidad de hacer más eficiente la técnicas de búsqueda local modificando la lista candidata y como consecuencia hacer más eficiente el espacio de búsqueda.

Y como último aspecto se tiene que el uso de las técnicas exactas conlleva entregar una solución óptima en tiempos impracticables.

A manera de conclusión se puede decir que tanto las técnicas de solución exactas como las técnicas de solución heurísticas tienen "*deficiencias*", ya que si se quiere obtener la

solución óptima es necesario contar con el tiempo suficiente para conocer la respuesta y por otro lado puedo obtener la respuesta en un tiempo eficiente pero sin tener la solución óptima. Las técnicas híbridas han resultado ser la mejor opción para dar respuesta muy cercana a la óptima en un tiempo razonable.

Siendo específicos las "*deficiencias*" de las técnicas heurísticas de construcción consisten en arrojar tours con una solución poco prometedora, y esto se debe, a que el tour tiene consigo cruces. Esta "deficiencia puede ser mejorada con las técnicas de búsqueda local y mejor aún, pueden desarrollarse nuevas técnicas híbridas que incorporen elementos de la geometría computacional que se valen de las propiedades "*naturales*" del problema, para obtener resultados muy cercanos al óptimo en un tiempo muy eficiente.

Dada esta situación los algoritmos que resuelven de forma óptima este problema, acarrean de manera inherente la inversión en tiempo para obtener una respuesta óptima en un tiempo razonable. Otra forma de dar solución a este problema es mediante la aplicación de las técnicas heurísticas, su uso ha permitido encontrar muy buenas soluciones en tiempos logarítmicos o bien polinomiales. Hoy en día las técnicas heurísticas, así como las técnicas híbridas de una técnica de optimización como la programación lineal con alguna técnica heurística han mostrado un gran desempeño en la solución a este problema [3]. De ahí que en este proyecto se aborden las técnicas de solución híbridas con elementos de la geometría computacional como medio para resolver este problema

1.6 Justificación

La técnica *2-Opt* es una de las técnicas de búsqueda local más básicas, exitosas y comúnmente usadas en el rubro de intercambio de aristas que resuelven el PAVE. En cada paso del algoritmo *2-Opt* se seleccionan dos aristas $e_1 = \{u_1, u_2\}$ y $e_2 = \{v_1, v_2\}$ del circuito de tal manera que u_1, u_2, v_1, v_2 sean distintos y aparezcan en este orden en el circuito y el algoritmo reemplaza estas aristas por las aristas $\{u_1, v_1\}$ y $\{u_2, v_2\}$, dando como resultado un circuito de menor longitud. El algoritmo termina en un óptimo local en el cual no se pueden lograr más mejoras. *2-Opt* puede lograr resultados sorprendentes en algunas instancias, tanto en tiempo como en cuanto a desviación con respecto a la longitud óptima, en un número sub cuadrático de pasos. [14]. Sin embargo, se han construido instancias para las cuales se requiere un número exponencial [158].

A través de un análisis de la literatura, se puede verificar que si dos aristas e**1** y e2 son aristas sucesivas, no es posible construir un circuito factible a través del intercambio descrito [4]. Por lo tanto, el único movimiento factible es el reemplazo de dos aristas no sucesivas e_1 y e_2. Resulta obvio que no se puede intercambiar una sola arista. Lo anterior implica que el tamaño de la vecindad de un circuito τ tiene $1+ n(n-3)/2 = \Theta(n^2)$ posibles circuitos o vecinos en total. Nótese que si la matriz de distancias no es simétrica, se tendría que seleccionar la dirección del circuito, lo cual implica que el tamaño de la vecindad

incrementa en un factor de dos 2+n(n-3). Así que, en una iteración de *2-Opt*, en el peor de los casos se tiene una complejidad de $O(n^2)$.

En este proyecto se propone reducir el conjunto de soluciones identificando solamente un subconjunto que es altamente prometedor, en lugar de tomar dos aristas de manera arbitraria, se toman dos aristas que se cruzan, y en cada una de las iteraciones se identifica este conjunto de aristas removiendo únicamente los pares de aristas que se intersecan y reconectando los caminos resultantes de tal manera que se mantenga un circuito hamiltoniano.

1.7 Metodología propuesta

La metodología completa consiste en: construir un circuito con alguna técnica de construcción o bien de manera aleatoria, en este proyecto se consideraron cinco técnicas de construcción: vecino más cercano y cuatro técnicas de inserción y que fueron afinadas con el elemento de geometría computacional, casco convexo. Una vez construido el circuito se procede a identificar las aristas que se cruzan, se guardan en una lista L y la técnica *2-Opt* toma de esta lista los pares de aristas que se están cruzando. Se consideraron tres formas de mandar llamar estas aristas: FIFO (que consiste en descruzar el primer par de aristas de la lista L, ALEATORIO (que toma cualquier par de aristas) y por último el PRIORIZADO (en el que previamente se identifica de la lista L, la arista de mayor longitud y es la que se descruza primero). El algoritmo para cuando la lista L ya no contiene aristas que se están cruzando, lo que equivale a decir que el circuito hamiltoniano ya no contiene aristas que se cruzan, e implícitamente que se ha alcanzado un óptimo local.

Este proyecto plantea la identificación del tipo de cruces a través de la estructura *intersección* con el paradigma de *barrido de plano*, implementado por primera en lenguaje de programación java 5.

La variante propuesta fue probada en un conjunto de instancias de la TSPLIB, lo cual nos garantiza saber con exactitud qué porcentaje de desviación con respecto al óptimo tiene el algoritmo híbrido propuesto.

1.8 Objetivo general

Proponer una metodología eficiente en tiempo y calidad de solución que resuelve el problema del agente viajero euclideano, que consiste en modificar el espacio de búsqueda de la técnica de búsqueda local *2-Opt* , así como la mejora de las técnicas de construcción por inserción, a través de la inclusión de elementos de Geometría Computacional, en instancias de la TSPLIB.

1.9 Objetivos específicos

- Analizar la taxonomía del problema del agente viajero general y euclideano para detectar las propiedades naturales con las que cuenta, y facilitar la inclusión de los elementos de la geometría computacional.
- Realizar un análisis de los algoritmos más frecuentemente utilizados que resuelven el problema del agente viajero general y en especial aquéllos que resuelven el problema del agente viajero euclideano para determinar los más eficientes hasta el momento en calidad del tour y tiempo de solución.
- Analizar las propiedades geométricas que se estudian en la Geometría Computacional.
- Incorporar las mejoras a los algoritmos previos (hibridar) y determinar la eficiencia computacional.
- Resolver instancias de la TSPLIB con los algoritmos híbridos propuestos y mostrar las estadísticas de desempeño.

Capítulo 2

Marco Conceptual

2.1 El problema del agente viajero y su historia

- **2.1.1 Definición y descripción del PAV como:**
  - **2.1.1.1 problema de programación lineal entero binario**
  - **2.1.1.2 asignación**
  - **2.1.1.3 asimétrico y simétrico**
  - **2.1.1.4 gráfico**
  - **2.1.1.5 métrico y no métrico**
- **2.1.2 El PAV como un problema combinatorio**
  - **2.1.2.1 Descripción del problema combinatorio**
  - **2.1.2.2 Planteamiento del PAV como un problema de optimización combinatoria (permutaciones)**
- **2.1.3 Problemas relacionados y variantes del PAV**
  - **2.1.3.1 El PAV y el problema de asignación cuadrático**
  - **2.1.3.2 El PAV y el problema de la ruta más larga**
  - **2.1.3.3 El PAV y los árboles de expansión mínimos**
- **2.1.4 Transformaciones simples del PAV**
  - **2.1.4.1 El cuello de botella del PAV**
  - **2.1.4.2 El PAV con ventanas de tiempo**
  - **2.1.4.3 El PAV con dependencia horaria**
- **2.1.5 Casos especiales**
  - **2.1.5.1 El Problema del agente viajero euclidiano**

2.2 El PAV y la complejidad computacional

- **2.2.1 Teoría de NP-Completez**
- **2.2.2 Buenos y malos algoritmos**
- **2.2.3 Problemas fáciles y difíciles**
- **2.2.4 El PAV es NP-Completo**
  - **2.2.4.1 El PAV como problema de decisión**

2.3 Tabla resumen de las instancias resueltas de manera óptima hasta el momento

---

2.1 El problema del Agente Viajero y su historia

Los orígenes del Problema del Agente Viajero datan de finales del siglo IX. Existen lecturas de referencia obligada cuando se quiere hablar de su historia como por ejemplo el de Lawler (1985) [1] y el de Lodi y Punnen (2002) [186]. En estos se destacan también las bases teóricas y experimentales respecto su planteamiento y resolución. El PAV ha atraído tanto la atención en diversos campos que ha provocado cientos de publicaciones y en la red existen cientos de sitios que hablan respecto a los orígenes, planteamiento y resolución de este fascinante problema.

2.1.1 Definición y descripción del PAV

El problema del agente viajero consiste en que un viajero desea visitar n ciudades, sin visitar a ninguna en más de dos ocasiones y regresando a la ciudad de la que partió (circuito hamiltoniano). Conoce el costo de viajar entre cada par de ellas (gráfica conexa) y debe planear su itinerario de tal manera que el costo total de su viaje sea el mínimo.

Este problema se puede formular como un problema de programación lineal, combinatorio (permutación), gráfico o de decisión. Esto se debe a los diversos campos que permiten abordar a este problema.
A continuación se describen estos planteamientos del PAV.

2.1.1.1 Problema de programación lineal entero binario

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 [4]

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

Donde x es un vector de variables de decisión, y $f(x), g_i(x)$ y $h_i(x)$ son funciones generales.
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 aquéllos con variables discretas, los últimos conocidos como "problemas combinatorios". [4]

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

$$sa$$

$$\sum_{j=1}^{n} a_{ij} x_j \geq b_i \qquad \forall_i = 1,\ldots,m$$

$$x_j \geq 0 \qquad \forall_j = 1,\ldots,n$$

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 a_{ij} $\quad i = 1,\ldots.,m \quad y \; j = 1,\ldots,n.$

Si atendemos a la estructura de estos problemas, el problema del agente viajero puede plantearse como un problema de programación lineal, y en especial entero, ya que el agente viajero no se puede quedar a mitad de camino, ni visitar media ciudad, o se visita o no se visita, o llega a la ciudad o no llega. Lo cual nos hace pensar en una solución entera. En donde lo que se quiere es minimizar (costo o tiempo) de todas las rutas posibles que puede tomar el agente viajero y las restricciones serían el que tiene que recorrer cada una de las ciudades sin poder visitarlas más de una vez (restricción 1) y que de la ciudad de la cual partió tiene que regresar (restricción 2).

Sin embargo, una característica especial que tiene el PAV es que no se permiten "subtours" (un subconjunto de ciudades), ya que el problema nos dice que el viajero desea visitar todas las ciudades y pasar por cada una de ellas una sola vez. Por lo que se tiene que agregar una tercera restricción "no se permiten subtours".

A continuación se describe una formulación general del PAV propuesta por [27] y se describe también en [1].

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 $N = \{1,\ldots,n\}$ 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

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

O con la siguiente fórmula: $\quad \sum_{i \in S} \sum_{j \in S} x_{ij} \geq 1$

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.

2.1.1.2 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.

2.1.1.3 El problema del agente viajero asimétrico y simétrico

En el problema simétrico la distancia entre dos ciudades es la misma independientemente de la dirección, formando una gráfica no dirigida. Esta simetría disminuye el espacio de soluciones a la mitad. Y en el caso asimétrico, las rutas no existen en ambas direcciones o bien las distancias son diferentes formando una gráfica dirigida. Las calles de un solo sentido o bien el pasaje aéreo de distintas ciudades con distintas tasas de llegada y salida son ejemplos de que la simetría puede resultar insuficiente.

De acuerdo a lo anterior, el PAV se puede plantear de la siguiente manera:

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

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

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

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

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

El problema del Agente Viajero Simétrico (*PAVS*)

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

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

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

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

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

De hecho el *PAVS* es un caso especial del Problema del Agente Viajero Asimétrico en el cual $c_{i\,j} \neq c_{ji} \quad \forall i, j \in V$. En este trabajo se trabajará con el *PAVS*.

2.1.1.4 Como un problema gráfico

El PAV también puede ser modelado como una gráfica, aquí las ciudades quedan representadas por los vértices, así como las aristas representan las rutas de una ciudad a otra y su longitud representa la distancia entre cada par de vértices de la gráfica. La solución al

PAV está dada por un circuito Hamiltoniano y la ruta óptima es el circuito Hamiltoniano más corto. Frecuentemente el modelo es una gráfica completa (esto es, una arista conecta a cada par de vértices). Si no existe una ruta entre dos ciudades (vértices), se agrega de manera arbitraria una arista de gran longitud que complete la gráfica sin afectar la solución final del *tour*.

El planteamiento formal del problema es: "Dada una gráfica completa y ponderada $G=(N,E,d)$ 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.

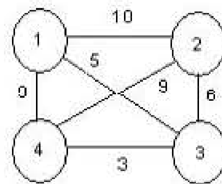


Figura 2.1 Ejemplo del problema del Agente Viajero como un grafo

2.1.1.5 El PAV métrico y no métrico

En el PAV métrico las distancias entre las ciudades satisfacen la desigualdad del triángulo. Esto puede entenderse como la distancia más corta entre la ciudad i y la j nunca va a ser más larga que la distancia pasando por la ciudad k.

$$c_{ij} \leq c_{ik} + c_{kj}$$

Estas longitudes de las aristas definen una métrica sobre un conjunto de vértices. Cuando las ciudades son vistas como puntos en el plano, la distancia entre cada par de ciudades se define mediante alguna función de distancia.

- En el PAV euclidiano la distancia entre dos ciudades es la distancia euclidiana.
- En el PAV rectilíneo la distancia entre cada par de ciudades es la suma de la diferencia de las coordenadas x e y. Esta métrica es llamada la distancia Manhattan o la métrica *city-block*.
- La métrica maxima, es la distancia entre cada dos ciudades que se define como el maximo de la diferencia de las coordenadas x e y.

EL PAV No métrico

El PAV que no cumpla la desigualdad del triángulo se dice que es no métrico. Muchos problemas que no cumplen con esta desigualdad se presentan en los problemas de ruteo, por ejemplo, viajar por avión puede ser más rápido aún cuando se está viajando por la distancia más larga.

2.1.2 El PAV como un problema de combinatorio

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 [17].

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. [18]

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![17]

2.1.2.1 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, pues se puede tener un procedimiento que resulte adecuado para resolver problemas pequeños o medianos, pero resultar impracticables cuando el tamaño del problema es grande. [17]

Una *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. [17]

Un problema de *Optimización Combinatoria* es un conjunto I de instancias que pueden variar en sus parámetros, por ejemplo, para el TSP se pueden variar el número de ciudades lo que la hace diferente a otra instancia.

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.

2.1.2.2 Planteamiento del PAV como un problema de optimización combinatoria (permutaciones)

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 $\pi^n = k \ \ \forall k$. 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 $|S| = (n-1)!$.

2.1.3 Problemas relacionados y variantes del PAV

Como se ha mencionado en varios apartados, el *PAV* tiene muchas variantes, entre las cuales se pueden señalar, el Problema del Agente Viajero con Ventanas de Tiempo *PAVVT*, con cuello de botella, con dependencia horaria o bien está relacionado con otros problemas como lo es el problema de asignación cuadrático. Para conocer un poco más acerca de estas relaciones y variantes se recomienda leer [1], [3], [4], [5], [19] y [W3].

2.1.3.1 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 difícil 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 instalación i en la localidad k y la instalación 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 instalaciones.

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 facilidades h y k, es decir la cantidad de interacción (tráfico) existente entre las facilidades; $C = \{c_{hi}\}$ es el costo de asignar la facilidad h en la localidad i. [W6]
Para n ciudades con $D = [d_{ij}]$ la matriz de distancias

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

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

$$\begin{aligned} &\min z \sum_{i,j,p,q=1}^{n} d_{ijpq} x_{ip} x_{jp} \\ &s.a. \\ &\sum_{p=1}^{n} x_{ip} = 1 && i = 1,\dots,n \\ &\sum_{i=1}^{n} x_{ip} = 1 && p = 1,\dots n \\ &x_{ip} \in \{0,1\} && i,p = 1,\dots n \end{aligned}$$

Este problema fue enunciado originalmente de una manera ligeramente menos general que la anterior por Koopmans y Beckmann (1957), como un problema de localización de plantas. En este problema, $d_{ijpq} = c_{ij} t_{pq}$

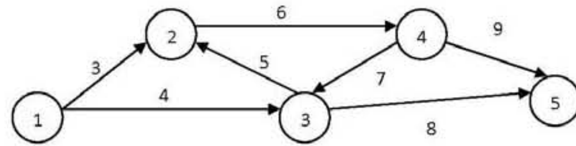
Donde t_{pq} es el número de artículos enviados de la planta *p* a la planta *q*, y c_{ij} el costo por enviar unidades de la localidad *i* a la localidad *j*.
La variable x_{ip} es un indicador de sí o no se asigna la planta *p* a la localidad *i*. Es importante hacer notar que la solución del *PAV* puede ser una permutación cíclica de los enteros $1,...,n$. Entonces x_{ip} puede ser interpretado como un indicador de si 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 $p=1,...n-1$, $t_{n1}=1$ y $t_{pq}=0$ de otra manera. [1]

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. [W6]

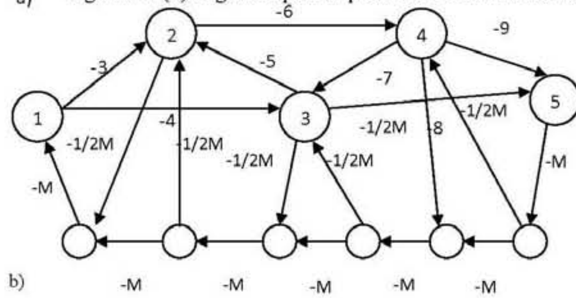
2.1.3.2 El PAV y el Problema de la ruta más larga

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 (difícil), 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 2.2(a).



a) Figura 2.2 (a) Digráfica para el problema de la ruta más larga



b)

Figura 2.2 (b) Transformación de la digráfica al TSP

Primero se multiplica cada duración del arco por (-1), para convertir el problema a 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 2.2 (b). Se afirma que el circuito hamiltoniano en el nuevo digrafo 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 $-2(n-2)$.

Cabe hacer la diferencia que el problema de la ruta más larga también conocida como PERT-CPM (Program Evaluation aand review Technique) y el CPM (Critical Path Method,) se diferencia del PAV ya que la metodología PERT muestra el camino como una secuencia de actividades conectadas, que conduce del principio del proyecto al final del mismo, por lo que aquel camino que requiera el mayor trabajo, es decir, el camino más largo dentro de la red, viene siendo la ruta crítica o el camino crítico de la red del proyecto.

Esto significa que el camino final puede implicar la visita de una actividad en más de una ocasión lo que viola la restricción del PAV. Tan solo el hecho de que la ruta critica no contempla regresar a la actividad de origen.

2.1.3.3 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.

2.1.4 Transformaciones simples del *PAV*

2.1.4.1 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 recorre, en lugar de minimizar la suma de las distancias de todos los recorridos. [1]

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 es apropiado.

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 $1 \leq i < m$ y tal que $d\ c_{\pi(m)}, c_{\pi(1)} \leq B$?

2.1.4.2 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 forma que el circuito debe abandonar cada vértice dentro de su ventana de tiempo y que 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 $a_i t_{j,i}$ $j \neq i$.

El *PAVVT* tiene importantes aplicaciones, especialmente en problemas de distribución y secuenciación.

2.1.4.3 El PAV con dependencia horaria (*PAVDH*)

En los problemas reales de rutas de vehículos, como 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 atascos 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 costo 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 costo 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 qué $a_i,b_i \in Z^+ \cup \{0\}$ y $[a_i,b_i] \subseteq [a_0,b_0]$ $\forall i \in \{1,...,n\}$. Consideramos para cada ventana de tiempo $[a_i,b_i]$ $p_i = b_i - a_i + 1$ periodos 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 periodo T_i^k
El *PAVDH* consiste en encontrar un circuito hamiltoniano en *G*, empezando y acabando 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.

2.1.5 Casos especiales

2.1.5.1 El Problema del Agente Viajero Euclidiano *PAVE*

Una restricción "natural" del problema del agente viajero es la desigualdad del triángulo, esto es: para tres ciudades A, B y C, la distancia entre A y B debe ser a lo más la distancia de A a B más la distancia de B a C. Es decir se cumple lo siguiente: $c_{AB} + c_{BC} \geq c_{AC}$

Cuando la distancia está dada por la distancia euclidiana, $d=\sqrt{(x_i-x_j)^2+(y_i-y_j)^2}$ ya que las distancias en el plano cumplen la desigualdad del triángulo, entonces estaremos hablando del Problema del Agente Viajero Euclidiano *PAVE*. Y aunque pudiera pensarse que es más "fácil" que el *PAV* general aún sigue siendo *NP-Hard*.

Definición del *PAVE*

Dados *n* nodos en el plano, se quiere encontrar un tour cerrado que visita cada nodo exactamente una vez y que se incurra en tener un mínimo costo. La suma del tour, es la suma de las distancias a lo largo del tour, y la distancia entre cada par de nodos está dada por la distancia euclidiana.

Para este conjunto de instancias en particular existen algoritmos de aproximación con una constante de aproximación, como el elaborado por Christofides en 1976 [20], el cual encuentra un tour de longitud de 1.5 veces del tour óptimo. También existe otro algoritmo que se aleja a lo más en el doble de longitud del tour óptimo. Otro algoritmo cuya aproximación al tour óptimo esta dado por el factor $(1 + 1/c)$ y $c>0$ en un tiempo $O(n (\log n)$ ^$O(c))$, diseñado por Sanjeev Arora [21]. La descripción de estos algoritmos se llevará a cabo en los siguientes capítulos.

2.2 El PAV y la 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. [17]

El *PAV* pertenece a la clase de problemas de optimización combinatoria que se conocen *NP-Completos*. Esto es, si alguien pudiera encontrar un algoritmo eficiente (por ejemplo uno que garantizara encontrar la solución óptima en un número polinomial de pasos) para el *PAV*, entonces los algoritmos eficientes podrían encontrar solución en tiempo polinomial para todos los problemas de la clase *NP-Completos*. Hasta el momento, no se ha encontrado un algoritmo de tiempo polinomial que resuelva el *PAV* de manera óptima.

Esto significa que ¿es imposible resolver cualquier instancia grande de estos problemas? No, muchos problemas de optimización prácticos verdaderamente de gran escala han sido resueltos de manera óptima. En 1994, Applegate, *et. al.* Resolvieron el *PAV* el cual modelaba la producción de tarjetas perforadas con 7,397 hoyos (ciudades), y, en 1998, el mismo autor resolvió el problema con 13,509 ciudades de Estados Unidos *E.U.* Y en el 2004, este mismo equipo resuelve una instancia con 24,978 ciudades.

2.2.1 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.

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.

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 $1,2,...,n$ y para cada par de ciudades i,j la distancia $c_{i,j}$ entre ellas. La solución es una permutación $(\pi_1,\pi_2,...,\pi_n)$ de ciudades que minimicen $\sum_{i=1}^{n-1} c_{\pi_i\pi_{i+1}} + c_{\pi_n\pi_1}$.

Un *ejemplo* de un problema se obtiene al especificar los valores de todos los parámetros del problema. Por ejemplo, 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.

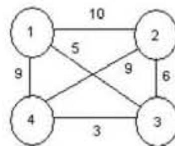


Figura 2.3 E*jemplo* del problema del agente viajero

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.

2.2.2 Buenos y malos algoritmos

Los algoritmos han sido divididos como buenos o malos. La comunidad computacional acepta que un buen algoritmo es aquél para el cual existe un algoritmo polinomial determinístico 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 puede resultar 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.

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 puede ser solucionada por un algoritmo polinomial 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 este puede resolver (ver tabla 1.1).

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.

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

2.2.3 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 bastante el tamaño de problemas que pueden resolverse en las computadoras modernas. Técnicamente hablando, 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. [22]

Una forma de obtener una solución al problema del agente viajero, como parte de un problema combinatorio, 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 $f(n)=(n)!$. [22]

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 en un tiempo de ejecución 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 1.2 tomada de [23], ilustra las diferencias de crecimiento de algunas 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 min
30	.00003 seg	.0009 seg	.027 seg	24.3 seg	17.9 mi	6.5 años
40	.00004 seg	.0016 seg	.064 seg	1.7 min	12.7 dias	3857 siglos
50	.00005 seg	.0025 seg	.125 seg	5.2 min	35.7 años	$2 \text{ X } 10^8$ años
60	.00006 seg	.0036 seg	.216 seg	13 min	366 siglos	$1.3 \text{ X } 10^{13}$ siglos

Tabla 2.2 Comparación de varias funciones polionomiales y exponenciales

2.2.4 El PAV es NP-Completo

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 *si* o *no*. El objetivo de realizar esta operación 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:

2.2.4.1 El PAV como problema de decisión

Si se plantea como un problema de decisión queda de la siguiente forma [3].

Instancia: Sea un conjunto C de m ciudades, la distancia para cada par de $c_i, c_j \in C$ ciudades y un número entero positivo B.

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

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

Se introducirá el concepto de reducción polinomial. Supóngase que se tiene un problema π_1 *que puede ser resuelto 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*. [23]

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:

¿No, 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 para 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 realiza 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.

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

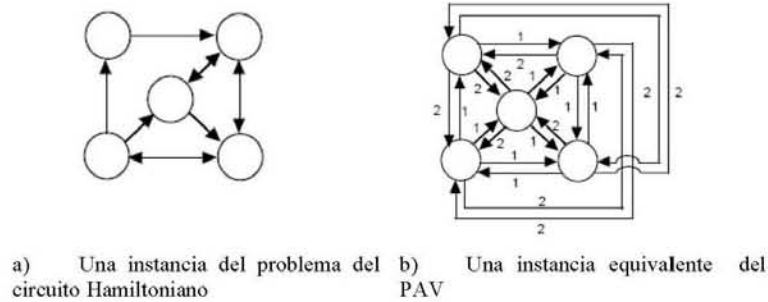
a) Tiene que ser un problema NP, es decir, mostrar que el PAV ($\prod$) pertenece a NP.
b) Se tiene que encontrar un problema $\prod'$ que se conoce es NP-Completo tal que $\prod'$ sea polinomicamente reducible al PAV ($\prod$).

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 hay así una solución para el PAV de costo *n.* En la figura 2.4 (b) se presenta la instancia PAV correspondiendo a la instancia HC de la figura 2.4 (a).

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 polinómicamente reducible al problema del agente viajero, por lo cual, se demuestra que el Problema del Agente Viajero es NP-Completo.



a) Una instancia del problema del circuito Hamiltoniano

b) Una instancia equivalente del PAV

Figura 2.4 Reducción polinomial del problema del circuito hamiltoniano al problema del agente viajero (PAV)

Como se ha comentado en párrafos anteriores, 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.

2.3 Tabla resumen de las instancias resueltas de manera optima hasta el momento

Los códigos de los programas de computadora para resolver las instancias del *PAV* han crecido de manera impresionante como se observa en la tabla 2.3. Comenzando por una instancia de 49 ciudades de Dantzig, Fulkerson, y Jonson en 1954 hasta la instancia resuelta en el 2004 (50 años después) de 24,978 ciudades.

Año	Equipo de investigadores	Tamaño de instancia	Nombre
1954	G. Dantzig, R. Fulkerson, y S. Johnson	49 ciudades	dantzig42
1971	M. Held y R.M. Karp	64 ciudades	64 random
1975	P.M. Camerini, L. Fratta, y F. Maffioli	67 ciudades	67 random
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, y W. Cook	13,509 ciudades	usa13509
2001	D. Applegate, R. Bixby, V. Chvátal, y W. Cook	15,112 ciudades	d15112
2004	D. Applegate, R. Bixby, V. Chvátal, W. Cook, y K. Helsgaun	24,978 ciudades	sw24798

Tabla 2.3 Resumen de las instancias resueltas de manera óptima.

Capítulo 3

El problema del agente viajero euclidiano

3.1 Definición y formulación matemática del PAVE
3.2 Fundamentos de las técnicas de solución heurísticas
3.2.1 Técnicas de Construcción
3.2.1.1 Estudios comparativos de los métodos constructivos
3.3 Algoritmos de búsqueda local en el problema del agente viajero euclidiano
3.3.1 El algoritmo 2-Opt
3.4 Algoritmos y esquemas de aproximación- α para el *PAVE*
3.5 Esquemas de aproximación
3.5 Técnicas Hiperheuristicas
3.6 El PAVE y la psicología

3.1 Definición y formulación matemática del PAVE

Los primeros documentos que mencionan el PAVE datan de los años 40´s, cuando el famoso estadista Mahalanobis [41] trabaja con esta versión del problema en donde discute cómo elegir de manera aleatoria algunas localidades ubicadas en el plano euclidiano. Este trabajo estuvo en conexión con un estudio de granjeros realizado en Bengal en 1938 en donde unos de los principales costos eran el transporte de hombres y equipo de un punto a otro. Aparentemente este trabajo es independiente al realizado por Merill Flood quien realiza algo similar pero unos años antes.

Dados n nodos en plano, se quiere encontrar un tour cerrado que visita cada nodo exactamente una vez y que se incurra en tener un mínimo costo. La suma del tour, es la suma de las distancias a lo largo del tour, y la distancia entre cada par de nodos está dada por la distancia euclidiana.

Englert [158] proporciona una definición formal del problema del agente viajero euclidiano simétrico.
Una instancia del PAV consiste de un conjunto $V = \{v_1, \ldots, v_n\}$ de *vértices* (dependiendo del contexto, también pueden ser llamados *puntos*) y una función de distancia d: $V \times V \rightarrow \Re_{\geq 0} \geq 0$ que asocia a cada par $\{v_i, v_j\}$ de vértices distintos una distancia $d(v_i, v_j) = d(v_j, v_i)$. La meta es encontrar un circuito de mínima longitud que visita cada vértice exactamente una vez y regresa al vértice inicial al final del recorrido. En este sentido el término tour se denota también como ciclo Hamiltoniano.
Un par (V, d) de un conjunto no vacío V y una función d: $V \times V \rightarrow \Re_{\geq 0}$ es llamado un *espacio métrico* si para todas las $x, y, z \in V$ se satisfacen las siguientes propiedades:

a) $d(x, y) = 0$ si y solo si $x = y$ (*reflexividad*),
b) $d(x, y) = d(y, x)$ (*simetría*),
c) $d(x, z) \leq d(x, y) + d(y, z)$ (*desigualdad del triángulo*)

En la figura 3.1 se puede observar gráficamente la propiedad de la desigualdad del triángulo.

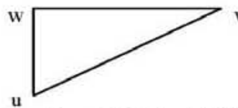


Figura 3.1 Desigualdad del triángulo

Si (V, d) es un espacio métrico, entonces d es llamado una *métrica* sobre V. Una instancia del PAV con vértices V y una función de distancia d es una *instancia métrica* del PAV si (V, d) es un espacio métrico.

Una clase de métricas bien conocidas sobre $\Re^d$ es la clase de métricas L_p. Para cada $p \in \mathbb{N}$, la distancia $d_p(x, y)$ de dos puntos $x \in \mathbb{R}^d$ y $y \in \mathbb{R}^d$ con respecto a la métrica L_p está dada por

$d_p(x,y) = \sqrt[p]{|x_1 - y_1|^p + \ldots + |x_d - y_d|^p}$. La métrica L_1 es llamada frecuentemente *métrica Manhattan* y L_2 es conocida como la *métrica Euclidiana*. Para $p \to \infty$, la métrica L_p converge a la métrica L_∞ definida por la distancia $d_\infty(x, y) = \max\{|x_1 - y_1|, \ldots, |x_d - y_d|\}$. Una instancia del PAV (V,d) con $V \subseteq \mathbb{R}^d$ en donde d es igual a d_p restringido a V es llamada una instancia L_p. También se utilizan los términos *instancia Manhattan* e *instancia Euclidiana* para denotar las instancias L_1 y L_2 respectivamente. Cuando p es claro en el contexto, se escribe d en lugar de d_p. En este trabajo se estudiaron solamente instancias Euclidianas.
Aún cuando el PAVE es un caso especial del problema del agente viajero general sigue siendo NP-Completo! Y estrictamente hablando NP-Hard demostrado por Papadimitriou en 1976, [144]

3.2 Fundamentos de las técnicas de solución heurísticas

Para resolver al PAVE al igual que al PAV general, existen técnicas de solución ***exactas***, [3], [4] y [5] que dan una respuesta óptima al problema a costa del tiempo excesivo que se requiere o la infraestructura que lleva consigo para obtener una respuesta única que se garantiza ser la óptima. Y dado que el PAVE sigue siendo NP-Completo, para salvar la dificultad computacional inherente a éste problema se ha inclinado por las técnicas heurísticas en combinación con elementos de la geometría computacional que nos arrojan una buena solución en un tiempo razonable, ya que las técnicas híbridas han mostrado obtener mejores resultados que las técnicas heurísticas propuestas exclusivamente para resolver este problema.

En la publicación del departamento de Estadística e Investigación Operativa de Rafael Martí, [24] y en [13] se definen y explican las técnicas heurísticas que resuelven el Problema del Agente Viajero Euclideano, para mayor referencia consultar las referencias señaladas con anterioridad.

Las diversas heurísticas que resuelven al PAVE se clasifican en términos generales en dos grande rubros, aquéllas que construyen una solución, a las que se les llama técnicas de construcción y aquéllas técnicas que mejoran ésta solución, a las que se les da el nombre de técnicas de búsqueda local o de mejora.

A continuación se muestra un resumen de éstas técnicas:

3.2.1 Técnicas de Construcción

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 y son los siguientes:

- Heurísticos del vecino más próximo o más cercano
- Heurísticos de Inserción

- Heurísticos basados en árboles generadores ó de expansión
- Algoritmo de Christofides
- Heurísticos Basados en Ahorros

3.2.1.1 Estudios comparativos de los métodos constructivos

En el trabajo desarrollado por Rafael Martí, se destaca un estudio comparativo del desempeño de las técnicas de construcción, que a continuación se muestra.

Heurísticas	Longitud obtenida	T. Ejecución
Vecino más cercano	$\frac{L(NN)}{L(OPT)} \leq (0.5)(\lceil \log_2 n \rceil + 1)$	$O(n^2)$
Inserción aleatoria	$\frac{L(Inc)}{L(OPT)} \prec 2 - \frac{2}{n}$	$O(n^2)$
Inserción más alejada	$\frac{L(Inc)}{L(OPT)} \prec 2 - \frac{2}{n}$	$O(n^2)$
Christofides	$L(H) \prec \frac{3}{2} L(OPT)$	$O(n^3)$
Ahorros	$\frac{L(Aho)}{L(OPT)} \leq \lceil \log_2 n \rceil + 1$	$\Theta(n^2 \log n)$

Tabla 3.1 Funciones teóricas promedio en tiempo de ejecución para métodos de construcción del PAV

Según la tabla 3.1 se puede observar que en el peor de los casos de todos los tiempos de ejecución teóricos están dados por funciones polinomiales. Estas técnicas muestran su bondad por tener estos tiempos, sin embargo pueden hacerse arreglos que pueden hacer más eficiente su desempeño, como con la técnica del vecino más cercano, que para reducir la miopía del algoritmo y aumentar su velocidad se considera un conjunto de vértices que son los más cercanos dentro de un subgrafo.

En este mismo trabajo, se muestran un conjunto de alternativas que nos permiten conocer el porcentaje de desviación con respecto al óptimo y se resumen a continuación:

- Comparación con la solución óptima
- Comparación con una cota
- Cota *Held-Karp*
- Comparación con un método exacto truncado
- Comparación con otros heurísticos
- Análisis del peor caso

En el caso del *PAVE* se cuenta con una cota diseñada por Held y Karp [2], que se utiliza cuando la solución óptima de los problemas estudiados no se conoce o bien cuando se generan instancias aleatorias. Existe una vasta literatura acerca de cómo mejorar esta cota como por ejemplo en [26] en donde se realizan algunas mejoras en el cálculo de esta cuota o límite inferior de la solución.

Para el caso particular de este proyecto existen un conjunto de instancias de dominio público TSPLIB [WWW1] para las cuales se conoce la cota óptima y se tomaron para medir el algoritmo híbrido propuesto.

3.3 Algoritmos de búsqueda local en el problema del agente viajero euclidiano

En este apartado se van a estudiar diversos algoritmos basados en la búsqueda local. Estas técnicas de acuerdo con Thomas Stützle [150] y Rafael Martí [24] y diversos estudios [186], son las que han resultado ser las más exitosas para resolver el PAVE.

Los algoritmos de búsqueda local, se pueden definir como: estrategias generales para resolver problemas de optimización sobre espacios de búsqueda exponenciales y se basan en explorar el entorno o vecindad de una solución. Utilizan una operación básica llamada movimiento o transformación como la definen otros autores, que, aplicada sobre los diferentes elementos de una solución, proporciona las soluciones de su entorno.

La idea básica de estos algoritmos es iniciar con una solución inicial generada de manera aleatoria o bien hallada con un algoritmo de construcción, se aplica una transformación o conjunto de transformaciones para mejorar la solución y ésta se convierte en la solución actual y se repiten estas acciones hasta que ninguna transformación del conjunto mejore a la solución actual, obteniéndose un óptimo local. Los procedimientos que implican un número de movimientos dinámicos en cada paso se llaman métodos de profundidad variable. Frecuentemente el éxito de estos procedimientos depende fuertemente de la calidad de la solución inicial.

Los progresos recientes en los procedimientos de búsqueda local están relacionados con el diseño de estructuras de vecindad más poderosas, estos avances se enfocan en estructuras de vecindad compuestas, es decir, realizan movimientos interdependientes en lugar de movimientos simples. Como regla general, cuanto más amplia sea la vecindad, mayor será tanto la calidad de las soluciones localmente óptimas como la precisión de la solución final que se obtenga. Pero, al mismo tiempo, cuanto más amplia sea la vecindad, más tiempo será necesario para realizar la búsqueda dentro de ella en cada iteración. Por esta razón, una vecindad muy amplia produce necesariamente una heurística más eficaz, a menos que la búsqueda se realice de un modo muy eficiente.

Los métodos de profundidad variable están basados en el intercambio de aristas e inserción de nodos, aquí se encuentran los procedimientos de k-intercambio o bien K-Opt y el de Or - inserción. El procedimiento de k-intercambio, donde k es estrictamente mayor a dos, se encuentran los procedimientos de 2-Opt, 3-Opt y hasta el momento solo se ha experimentado con 5 movimientos.

El procedimiento de vecindad 2-Opt o el algoritmo de 2-Opt, se puede generalizar para desempeñar k-opt movimientos que eliminan k aristas y se agregan nuevas k aristas. En total hay $\binom{n}{k}$ posibles formas de eliminar aristas del tour y hay $(k-1)!2^{k-1}$ formas de volver a unir al subgrafo desconectado, incluyendo el tour inicial para recuperar la estructura del tour. Para valores pequeños de *k*, relativo a *n*, esto implica un tiempo de complejidad $O(n^k)$ para verificar la k-optimalidad.

El algoritmo de Lin-Kernighan o variable- Opt (V-Opt) [16], permite tener transformaciones o movimientos que empeoran el valor de la solución, lo que conduce a un espacio de búsqueda más complejo, utilizando movimientos de mejora como de no mejora, permitiendo no preservar la conectividad del tour.

Mientras el método *k-opt* remueve un número fijo *k* de aristas del tour original, el método Variable –Opt no fija el tamaño de las aristas a remover. En lugar de esto, incrementa el tamaño de *k* conforme el proceso de búsqueda avanza. El mejor método conocido de esta familia es el método Lin–Kernighan. Shen Lin y Brian Kernighan fueron los primeros en publicar su método en 1972, y fue el método más confiable para resolver el PAVE durante casi dos décadas. Se desarrollaron a finales de los 80´s métodos más avanzados desarrollados por Bell Labs por David Johnson y su equipo de investigación. Estos métodos (llamados Lin-Kernighan – Jhonson) se basan sobre el método Lin–Kernighan, agregando ideas de búsqueda tabu e ideas de computación evolutiva. Las técnicas básicas de Lin–Kernighan ofrecen resultados al menos como la que ofrece la técnica 3-opt. Los métodos Lin–Kernighan–Johnson calculan un tour Lin–Kernighan, y lo perturban con una mutación que remueve al menos cuatro aristas y se reconectan para formar un nuevo tour. La mutación es suficiente para mover el tour del mínimo local. Los métodos V-opt son ampliamente considerados como los métodos más poderosos para resolver el PAVE y otros problemas como circuito Hamiltoniano y otros casos del PAV como el no métrico para los cuales otras heurísticas fallan. Por muchos años el método Lin–Kernighan–Johnson ha identificado las soluciones óptimas para cuyos problemas se conoce la solución óptima y ha identificado las mejores soluciones para todos los PAV en los cuales ha sido probado. Dentro de esta familia se encuentra el algoritmo 2-Opt, quien ah resultado ser una de las técnicas más poderosas para resolver al PAVE.

Las cadenas de expulsión es una técnica de búsqueda local o por vecindad, en donde esta es "muy grande" con respecto al tamaño de los datos y en los que la búsqueda local se hace con criterios de máxima eficiencia. Hay tres tipos muy amplios de algoritmos de búsqueda por vecindad a muy gran escala (VLSN: Very Large-Scale Neighborhood): (1) métodos de profundidad variable en los que la búsqueda local se realiza de modo heurístico, (2) aplicación de la programación dinámica o de técnicas de flujo de redes a vecindades amplias, y (3) vecindades grandes inducidas por restricciones del problema original que pueden resolverse en tiempo polinómico.

Los algoritmos basados en esta filosofía son métodos de profundidad variable que generan una secuencia simple de movimientos interrelacionados para crear movimientos compuestos. Habitualmente las *Ejection Chains* están relacionadas con las restricciones; el término inglés "ejection" significa *expulsión, salida*, y alude a que al hacerse cambios en ciertos elementos se causa que otros elementos sean **expulsados** de su estado actual, debido a que en caso contrario se produciría una infactibilidad.
Cada *Ejection Chain* es una composición o cadena de movimientos simples, no necesariamente "mejoradores", como se observa en la figura 3.32 en donde la ejecución de los movimientos simples m1, m2 y m3, provocan un buen movimiento m4; a su vez la ejecución de los movimientos m1, m2 …, m5 provoca otro buen movimiento m6.
Existe una gran literatura que nos habla de las cadenas de expulsión, teniendo origen en [28] por Glover. Se recomienda [37] para un estudio más exhaustivo.

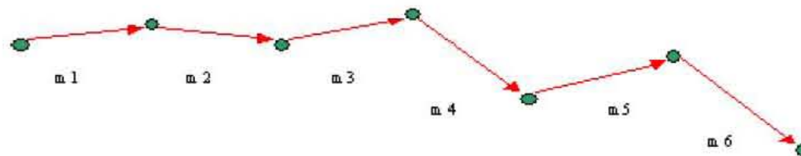


Figura 3.2 Ejection Chains

En la literatura no se muestra las funciones teóricas de las longitudes obtenidas a partir de esta estrategia. Sin embargo, se han hecho diversos estudios con otro enfoque en donde se observa que en tiempo polinómico, permiten realizar búsquedas de modo implícito en un subconjunto de tamaño exponencial de la vecindad de k-intercambio cuando k no es un valor fijo.

Es decir, las mejoras realizadas a este tipo de heurísticas se enfocan en el tamaño de la vecindad y el tiempo teórico para alcanzar las búsquedas de intercambios.

En [24] ,[186] se encuentra detallado un estudio de los algoritmos de búsqueda local que resuelven el Problema del Agente Viajero Eucidiano y se resumen a continuación.

Dentro del conjunto de procedimientos o algoritmos que han resultado ser los más poderosos para resolver el PAVE se destacan los siguientes:

- Algoritmo 2-Opt
- Algoritmo de k – intercambio o k-Opt
- Algoritmo de Lin y Kernighan
- Cadenas de expulsión

A continuación se detalla el algoritmo 2-Opt ya que es punto de investigación de este trabajo.

3.3.1 El algoritmo 2-Opt

Flood en 1956 [156] hace una observación para grafos con distancias euclídeas (o en general con costos cumpliendo la desigualdad triangular), mediante la cual se basa este algoritmo, 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.

La técnica 2-Opt es una de las técnicas de búsqueda local más básicas, exitosas y comúnmente usadas en el rubro de intercambio de aristas que resuelven el PAV simétrico. Propuesta por Croes en el año de 1952, [55]. Esta técnica comienza con un circuito inicial arbitrario o bien previamente construido con alguna técnica de construcción y se va mejorando este circuito sucesivamente, intercambiando dos de las aristas contenidas en el circuito con otras dos aristas. De manera más precisa, en cada paso del algoritmo 2-Opt se seleccionan dos aristas e1 = {u1, u2} y e2 = {v1, v2} del circuito de tal manera que u1, u2, v1, v2 sean distintos y aparezcan en este orden en el circuito y el algoritmo reemplaza estas aristas por las aristas {u1, v1} y {u2, v2}, dando como resultado un circuito de menor longitud. El algoritmo termina en un óptimo local en el cual no se pueden lograr más mejoras.

Se puede verificar que si dos aristas e1 y e2 son aristas sucesivas, no es posible construir un circuito factible a través del intercambio descrito. Por lo tanto, los únicos movimientos factibles son el reemplazo de cero aristas, o de dos aristas no sucesivas e1 y e2. Resulta obvio que no se puede intercambiar una sola arista. Lo anterior implica que el tamaño de la vecindad de un circuito τ tiene $1+ n(n-3)/2 = \Theta(n^2)$ posibles circuitos o vecinos en total. Nótese que si la matriz de distancias no es simétrica, se tendría que seleccionar la dirección del circuito, lo cual implica que el tamaño de la vecindad incrementa en un factor de dos $2+n(n-3)$. Así que, en una iteración de 2-Opt, en el peor de los casos se tiene una complejidad de $O(n^2)$.

La técnica 2-Opt puede lograr resultados sorprendentes en algunas instancias, tanto en tiempo como en calidad del tour con respecto a la longitud óptima, en un número

subcuadrático de pasos [26]. Sin embargo, se han construido instancias para las cuales se requiere un número exponencial [158].

Existen estudios experimentales [171], [160] que muestran cómo se pueden lograr mejoras importantes en tiempo de respuesta de la técnica 2-Opt. También se han desarrollado trabajos en los cuales se muestran implementaciones de hardware muy eficientes [161]. Sin embargo, los análisis teóricos siguen siendo limitados, [158].

Existen dos aspectos que se tienen que analizar cuando se implementa la técnica de búsqueda local 2-Opt. En primer lugar se debe corroborar si es más conveniente proponer un circuito inicial dado por una técnica de construcción del que se sabe que tiene un buen desempeño, o bien generarlo de manera aleatoria. Babin en 2005, [160] encontró que la técnica 2-Opt ofrece resultados significativamente mejores que una técnica de intercambio de cadenas, cuando se parte de soluciones generadas de manera aleatoria. Sin embargo, Perttunen en 1994, [162] mostró que el desempeño de las técnicas de intercambio de aristas, en especial la técnica 2-Opt, muestra mejores resultados cuando se parte de una solución inicial creada por una técnica de construcción con buen desempeño.

El otro aspecto a tomar en cuenta es de qué forma se selecciona el movimiento de mejora a realizar: puede ser el primer movimiento de mejora detectado en cada iteración o bien el que después de un cálculo de ahorro resulte ser el mejor movimiento de mejora. Estos temas fueron parcialmente contestados por Hansen y Mladenovic en el 2004, [188] quienes concluyen que para la técnica 2-Opt realizar el primer movimiento de mejora es ligeramente mejor y más rápida que seleccionar el mejor movimiento si se comienza con un circuito generado de manera arbitraria; se invierten los resultados cuando se parte de un circuito generado con alguna técnica de construcción.

Aunado a los dos aspectos mencionados en los párrafos anteriores, Johnson y McGeoch [189] describen cuatro reglas para ahorrar tiempo de corrida al implementar la técnica en un lenguaje de programación:

1) evitar las redundancias del espacio de búsqueda: cuando se implementa la técnica r-Opt, se puede ahorrar tiempo incluyendo un cálculo que detecte los intercambios que no mejoran la solución;

2) generar una lista de vecinos restringidos: únicamente se consideran los vecinos más cercanos de un vértice p para llevar a cabo las reconexiones (esta regla fue inicialmente propuesta por Zweig (1995);

3) generar una lista de "don't look bits" o movimientos que se sabe han sido fallidos en movimientos pasados;

4) representación del circuito a través de árboles: utilizar este tipo de representaciones del circuito acelera los tiempos de cómputo.

3.4 Algoritmos y esquemas de aproximación- α para el *PAVE*

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 $\varepsilon \succ 0$ qué: $c_h \leq (1+\varepsilon)c_{opt}$

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 $\Pi(con|I| = n)$ 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 Π 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 [3].

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 Π, con α una constante. Si A es un algoritmo de aproximación tal que para toda instancia I de Π, produce una solución que está dentro de α -veces el OPT(I).[13]

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. [13]

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.

En este sentido, si tenemos que un problema $\prod$ está en la clase APX (Problema de Aproximación X) si existe un algoritmo de tiempo polinomial para $\prod$ cuyo factor de aproximación o garantía está acotada por una constante [3].

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].
El PAVE como un subcaso del PAV métrico ocurre que el costo de las aristas satisface la desigualdad del triángulo (Δ PAV) existe un algoritmo simple que, siempre produce una ruta de longitud a lo mas dos veces la longitud de la ruta óptima. Esto tiene un tiempo de ejecución de $O(n^2)$, el cual se llama **duplicación** de aristas. También existe otro que se llama el algoritmo de **Christofides** con un tiempo de corrida de $O(n^3)$ siempre encuentra una ruta con constante de aproximación de a lo más 3/2 de longitud de la ruta óptima. Sahni y Gonzalez [151]) demuestran que el PAVE se puede aproximar al óptimo dentro de un factor constante. Goemans conjetura que la técnica Held-Karp tiene un radio de aproximación de 4/3 [152], sin embargo la mejor cota es la de 3/2 veces el optimo.
Karp [153] en un seminario sobre análisis probabilístico de los algoritmos demuestra que cuando n se selecciona de manera uniforme e independiente de un cuadrado unitario, la heurística de disección fija encuentra con alta probabilidad tours cuyo costo están dentro de un factor (1+1/c) del óptimo. Donde c es arbitrariamente largo.

3.5 Esquemas de aproximación

Arora en 1996, [21] diseña un esquema de aproximación para el PAVE y de manera independiente Mitchell [154] en el mismo año encuentra otro esquema de aproximación. El esquema de aproximación de Arora encuentra la solución óptima dentro de un factor (1+1/c) dentro de un tiempo $O(n(\log n)^{(O(\sqrt{d}c))^{d-1}})$.

Los esquemas de aproximación aún cuando ofrecen soluciones cercanas a las óptimas en un tiempo polinomial, son netamente teóricos y cuando se llevan a la práctica no suelen ser tan prometedores. De manera reciente Rodekel ,[155] realiza una implementación al esquema de aproximación de Arora y obtienen resultados no tan alentadores, ya que si bien los tiempos de ejecución son mejores que otros algoritmos la calidad del tour es muy pobre.

3.6 Técnicas Hiperheuristicas

Las metaheurísticas son estrategias inteligentes para diseñar o mejorar procedimientos heurísticos muy generales con un alto rendimiento. Entre las más populares y bien estudiadas técnicas se encuentran GRASP [30], Recocido Simulado [31], Búsquedas Tabú

[32], y los Algoritmos Genéticos [33]. Las cuales se han aplicado con éxito en la solución de un gran número de problemas reales.

Sin embargo, muchos de estos enfoques carecen de la robustez para resolver una amplia gama de problemas. Las metaheurísticas pueden resolver eficientemente un problema real luego de realizar un diseño de experimento profundo para ajustar sus parámetros, pero no pudiera resolver en lo absoluto, o da soluciones muy pobres, a otros casos incluso derivados de los mismos problemas [9]. El ajuste de parámetro de una misma metaheurística en la solución de varios problemas puede llevar mucho tiempo.

En los últimos años los investigadores de esta temática han propuesto una nueva técnica llamada hiperheurística [31 y 34]. La hiperheurística es un algoritmo de clase abstracta que opera a un nivel más alto que las metaheurísticas, y dirige a un grupo de heurísticas simples (de un nivel más bajo) las cuales son aplicadas, dependiendo de la característica del espacio de soluciones donde se encuentra explorando [31].

Una hiperheurística selecciona a cada paso la más prometedora heurística simple (o una combinación de heurísticas) que puede mejorar potencialmente la solución. Por otro lado, si no hay mejora, es decir, fue encontrada una solución óptima local, ella es capaz de diversificar la búsqueda a otras áreas del espacio de solución realizando una apropiada selección de un nuevo juego de heurísticas. Las heurísticas de bajo nivel, normalmente representan a los métodos de búsquedas locales o a las técnicas de construcción [31].

Todo el dominio de información se concentra en el conjunto de heurísticas simples y en la función objetivo. La selección de una nueva heurística está basada por distintos criterios: los valores obtenidos de la función objetivo, el tiempo de ejecución de la CPU, etc., siendo necesario para estos casos una perturbación de la solución [34].

Los enfoques hiperheurísticos con respecto a las metaheurísticas tienen las siguientes ventajas principales. Primeramente, una hiperheurística es sencilla y rápida de implementar; al mismo tiempo puede producir soluciones comparables en calidad a otras ya encontradas por eficientes metaheurísticas. Segunda, ella no tiene el acceso al conocimiento del dominio específico del problema, haciéndola aplicable en pequeños estudios o a problemas reales pobremente estudiados.

Finalmente, estas técnicas son robustas: pueden ser eficazmente aplicadas a una amplia gama de instancias de un problema.

3.7 El PAVE y la psicología

Cabe mencionar que estudios psicológicos han comprobado que el ser humano, de forma intuitiva, sabe que una condición necesaria para llegar a un circuito óptimo es la ausencia de cruces. Macgregor, [167], Graham, [168] y Rooij, [169] plantearon un problema de tipo TSP a un grupo de 1000 personas totalmente ajenos al campo de la investigación de operaciones y estratificados por edad y experiencia; encontraron que una sola persona incluyó un cruce en el circuito resultante, porqué esa fue la única forma de cerrar el circuito propuesto. Basado en pruebas estadísticas, los autores concluyeron que, independientemente de su experiencia, un sujeto sabe que para lograr un circuito óptimo hay que evitar cruces en la solución.

Capítulo 4

La Geometría Computacional y la solución del PAVE

4.1 Antecedentes
4.2 Geometría computacional
4.3 Optimización Combinatoria vs. Métodos Geométricos
4.4 Paradigmas y estructuras de dato geométricas
4.4.1 Paradigmas
4.4.2 Iterativo
4.4.3 Barrido del plano
4.4.4 Divide y vencerás
4.4.5 Ramificación y Acotamiento
4.5 Estructuras de datos geométricos
4.5.1 Cascos Convexos *CH(S)*
4.5.1.1 Definiciones
4.5.2 Algoritmos para calcular cascos convexos
4.5.2.1 Algoritmo Quick Hull
4.5.2.2 Algoritmo Scan de Graham
4.5.2.3 Algoritmo incremental
4.5.2.4 Algoritmo divide y vencerás
4.5.2.5 Algoritmo envoltura de regalo (Gift Wrapping)
4.6 Intersección de segmentos de línea
4.6.1 Algoritmo de Fuerza Bruta
4.6.2 Algoritmo de Barrido de Plano
4.6.2.1 Pseudocódigo del algoritmo Bentley-Ottman
4.6.2.2 Algoritmo Intersección de segmentos con barrido de plano O(n *log* n)

4.1 Antecedentes

La inclusión de los paradigmas y estructuras de datos de la Geometría Computacional *GC* en la técnica de búsqueda local 2-opt , se debe de manera fundamental a tres observaciones que se podrían considerar aisladas, sin embargo, guardan una lógica entre sí y coadyuvan a lograr una mejor eficiencia en los algoritmos propuestos, como objetivo principal de este proyecto.

Una de las primeras observaciones es que las técnicas de construcción como: vecino más cercano, y las técnicas de inserción con sus diferentes criterios, los algoritmos de aproximación como el de Christofides [20], devuelven una solución que es poco alentadora, se podría decir que tienen "*deficiencias*", debido a que la solución en el mejor de los casos tiene consigo al menos un cruce. Algunos estudios experimentales como los mostrados por Martí, [24] y los elaborados en este proyecto demuestran este hecho.

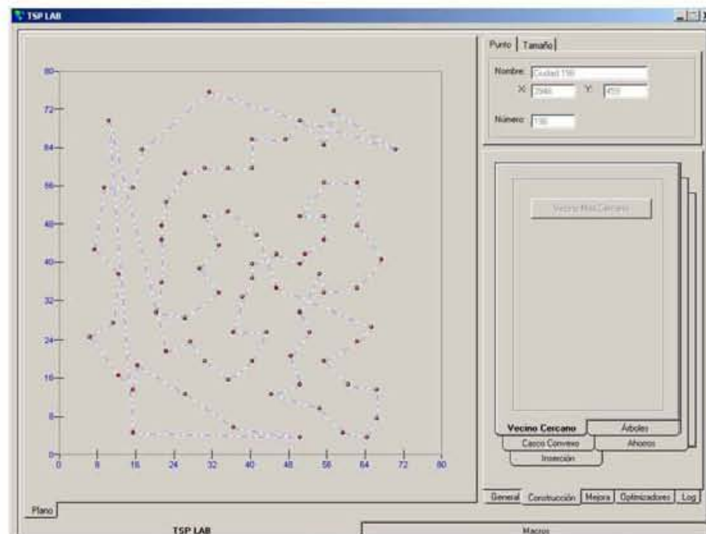


Figura 4.1. Instancia eil76 de la TSPLIB, solución final con cruces con técnica de vecino más cercano

Otra de las observaciones es que en el caso especial del PAVE que de manera "*natural*" cumple con la desigualdad del triángulo, lo cual quiere decir que siempre podremos acortar los tours dirigiéndonos de manera directa a una ciudad sin pasar por ciudades intermedias, y el hecho de que si un ciclo Hamiltoniano se cruza a sí mismo, puede ser fácilmente acortado [51], entonces basta con eliminar las dos aristas que se cruzan y reconectar los dos

caminos resultantes mediante aristas que no se corten, lo que da como resultado que el ciclo final es más corto que el inicial. Lo cual implica que siempre se puede mejorar un tour utilizando la propiedad "*natural*" del PAVE.

Como una tercera observación, se tiene que una de las técnicas de búsqueda local que "*invita*" de manera natural a "*quitar*" los cruces, es la técnica 2-opt. Ya que su estrategia de mejora consiste en tomar cualquier par de aristas, removerlas y reconectar el tour con otro par de aristas. Sin embargo, el espacio de búsqueda en una iteración es de $O(n^2)$, lo cual nos hace pensar que podemos reducirlo si nos fijamos únicamente en los pares de aristas que se cruzan.

Recientemente en el año 2008 Englert y otros investigadores [158] muestran uno de los pocos estudios teóricos en cuanto la calidad del tour y el número de iteraciones necesarias para entregar una solución libre de cruces. Los resultados tienen su lado positivo y otro negativo. Del lado negativo tenemos que existen un conjunto de instancias para las cuales se requiere un número exponencial de pasos para entregar un tour libre de cruces, del lado positivo, existen instancias que requieren un número polinomial de pasos. En este sentido, Leeuwen y Schoone , [164] analizaron una variante de esta técnica y demostraron que si en un circuito Hamiltoniano se reemplazan iterativamente solo las aristas que se intersecan , el circuito resultante se logra en un número de pasos en $O(n^3)$, lo cual significa una mejora sustancial en comparación con la complejidad exponencial que puede tener la técnica 2-Opt clásica. Este tiempo teórico incluye la posibilidad de generar en iteraciones intermedias cruces que no estaban en el tour original. Sin embargo el reporte no muestra la calidad del tour libre de cruces. Este estudio, logra conjeturas experimentales que muestran que la variante propuesta (tomando como pares de aristas aquellas que se intersecan en el tour original) arroja tours que se alejan como máximo en un 1.7% de las soluciones propuestas por la técnica 2-opt clásica. Existen otros estudios elaborados por Perttunen en 1994 [162] y Okano en 1999 [170] así como en este estudio, que conjeturan que la técnica 2-opt clásica logra un porcentaje de desviación con respecto al óptimo del 6% al 7% aproximadamente, lo cual significaría que la técnica propuesta estaría desviada del óptimo en un 8% en el peor de los casos.

Bentley en 1992, [157] comenzó a insertar conocimientos de la Geometría Computacional (GC) y muestra de manera experimental que algunas estructuras de datos geométricos en especial "*fixed radius near neighbor* "permiten un mejor desempeño en diversas heurísticas para resolver el problema del agente viajero euclideano, en especial la técnica 2-opt.

Algunas estructuras geométricas y paradigmas de la GC permiten un mejor desempeño en las técnicas heurísticas. En este proyecto se utiliza el paradigma de ***barrido de plano*** para identificar los cruces ente cada par de aristas y así obtener un espacio de soluciones

reducido y por tanto un menor número de iteraciones que la que ofrece la técnica 2-opt clásica. También se utiliza la estructura geométrica de ***casco convexo*** para que las técnicas de construcción lograran mayor eficiencia, ya que partiendo de una estructura geométrica las técnicas de construcción toman ventaja y entregan una mejor solución que ejecutando las técnicas heurísticas de construcción sin el apoyo de estos. Los paradigmas y estructuras geométricas utilizadas en la experimentación se describen en este capítulo.

4.2 Geometría computacional

La *Geometria Computacional* (*GC)* es una disciplina que se ocupa del diseño y análisis de algoritmos que resuelven problemas geométricos. El estudio de estos problemas tiene una historia ya trazada desde Euclides y más aún sus antecedentes pueden encontrarse en la Grecia clásica, hace 2600 años, donde los problemas geométricos que se planteaban los matemáticos de la época eran abordados desde el punto de vista constructivo, es decir algorítmico.[171]

Pero no es hasta 1975 en donde la *GC* surge como un campo en ciencias de la computación teórica. Preparata y Shamos en 1988, [174]. En un principio la *GC* tenía un énfasis especial en el desarrollo de algoritmos eficientes asintóticamente para problemas que contienen un gran número de objetos geométricos simples (puntos, líneas, planos, polígonos y poliedros).

El interés teórico de la *GC* ha sido suplementado por muchas aplicaciones prácticas como en robótica, procesamiento de imágenes, SIG (Sistemas de Información Geográfica), investigación de operaciones y muchos otros campos. [175]

Dado que existen problemas NP-Completos para los cuales la solución exacta están fuera de alcance, los paradigmas de la *GC* y las estructuras de datos geométricos han resultado ser muy usadas junto con el diseño de heurísticas prácticas para resolver éste tipo de problemas.

De manera reciente estos poderosos paradigmas y las estructuras de datos geométricas de la GC se han aplicado con gran éxito para algunos problemas de Diseño de Redes Topolológicas (*DRT*) en la rama de la investigación de operaciones.

Uno de estos ejemplos es el *Problema del Agente Viajero Euclideano* (*PAVE*) en donde la *GC* ha hecho posible encontrar soluciones muy cercanas a la óptima a instancias de problemas de cientos y miles de ciudades que pertenecen a un plano.[157]

Ni las técnicas clásicas de la programación matemática, ni las metodologías de la optimización combinatoria aplicadas a la solución del *PAVE* pueden competir con las técnicas tradicionales de la *GC*. [171]

Existen básicamente cinco áreas en la *GC* que tienen una importancia especial:

- Búsqueda y Ordenación: Localización de puntos, inclusiones convexas, inclusiones de polígonos, etc.
- Convexidad: Cascos convexos en E^2, E^3, E^d, estimación estadística, regresión, etc.
- Proximidad: Localización en general y búsqueda de vecindades, vecinos más cercanos, triangulaciones, solución de ecuaciones, etc.
- Intersección: Problemas de líneas ocultas, reconocimiento de patrones, programación lineal, etc.
- Optimización: Diseño de redes, árboles de expansión mínimos, agente viajero, cartero chino, programación lineal y programación entera, problemas de localización max-min y mini-max, etc.

La *GC* hace uso de muchos paradigmas y en especial el paradigma de "Divide y vencerás" así como el de estructuras de datos (listas, árboles binarios,...) y más aún estructuras tan especiales como los cascos convexos, las triangulaciones de Delaunay y los diagramas de Voronoi que juegan un papel crucial en la *GC*.

Existe literatura en la cual se afirma que los métodos de programación matemática y los de optimización combinatoria son inferiores a los métodos que ofrece la *GC* para resolver problemas de DRT, [171].

4.3 Optimización Combinatoria vs. Métodos Geométricos

Muchos de los problemas de la GC pueden ser formulados también como problemas de optimización combinatoria sobre gráficas de pesos. Como consecuencia estos pueden ser resueltos por medio de algoritmos de optimización de redes y heurísticas. En particular, la formulación de flujo de redes juega un papel muy importante en la formulación de problemas de DRT sobre gráficas con pesos. En general se puede decir que la geometría computacional y la optimización combinatoria son complementarias lejos de verse conflictuadas aunque los procedimientos utilizados en cada uno de los métodos son completamente diferentes.[171]

Un ejemplo típico de problemas de DRT que son resueltos por optimización combinatoria es el *Problema de Árboles de Expansión Mínimos Euclideanos* (PAEME) En donde Z-puntos son representados por vértices en una gráfica de pesos completa K_n y K_n contiene aristas entre cualquier par de vértices. Este problema puede ser resuelto en un tiempo $O(n^2)$ utilizando el algoritmo de Prim.

Otro problema clásico en donde los métodos de la *GC* han actuado de manera exitosa, es el *Problema del Agente Viajero Euclideano* (PAVE). Bentley [157] demuestra que las

estructuras de datos geométricos de la GC se han utilizado para acelerar muchas heurísticas para resolver este problema.

4.4 Paradigmas y estructuras de dato geométricas

Hay un gran número de paradigmas y estructuras geométricas dentro de la *GC* de vital importancia. A continuación se explican algunas de ellas y se detallan aquellas que se tomaron en cuenta para llevar a cabo la hibridación de la técnica 2-opt.

4.4.1 Paradigmas

Muchos de los paradigmas son frecuentemente utilizados por algoritmos y heurísticas que resuelven problemas de la GC.

A continuación se muestra un diagrama con los paradigmas más frecuentemente utilizados en la GC para resolver problemas relacionados con la geometría o bien relacionados con otras áreas.

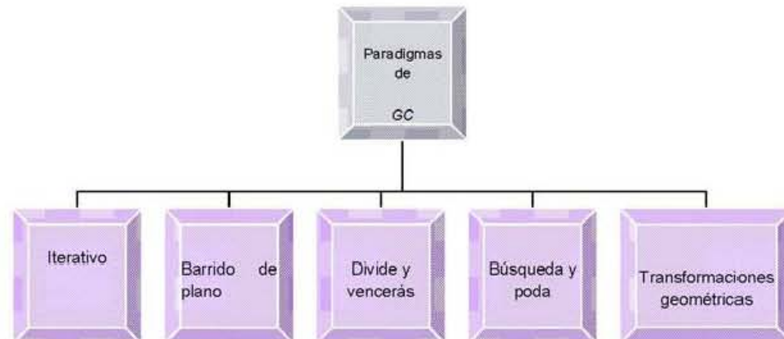


Figura 4.2. Árbol morfológico de los paradigmas de la *GC*

4.4.2 Iterativo

Las propiedades locales y las características de un problema dado, por ejemplo, la localización de puntos y la proximidad entre ellos, se utilizan para ir expandiendo o mejorando su solución iterativamente. En cada iteración se presentan varias opciones de cómo expandir la solución (o mejorar una solución factible). La opción greedy (glotona) da un incremento en costo muy baja y siempre es seleccionada.

El paradigma de iterativo es uno de los más básicos e intuitivo de los de este conjunto de paradigmas. Cuando se expande el conjunto de soluciones, el número de iteraciones es

proporcional al tamaño de instancia del problema. Cuando este paradigma se aplica a problemas NP-Completos, es usual obtener no tan buenas soluciones.

Cuando se mejoran las soluciones factibles, el número de iteraciones puede ser substancial ya que las mejoras continúan tanto tiempo como sea posible. Como un ejemplo de algoritmo que ilustra este paradigma es el del vecino más cercano, ya que en cada iteración el algoritmo va insertando la ciudad más cercana en distancia al conjunto de soluciones.

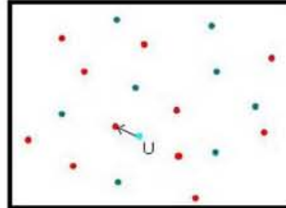


Figura 4.3 Técnica del vecino más cercano

4.4.3 Barrido del plano

El paradigma de barrido de plano es netamente geométrico. Existe un algoritmo cuya idea central es este paradigma, que consiste en mover una línea S a través del plano para detectar ciertas configuraciones de objetos geométricos (puntos, segmentos de líneas, rectángulos) que pertenecen al plano. Colecciona información de estos objetos en las secciones de cruce actualizando el status de S. Este status tienen muchos cambios pero finitos en posiciones estratégicas de S. Estas posiciones estratégicas se coleccionan sobre una línea eventual E perpendicular a S.

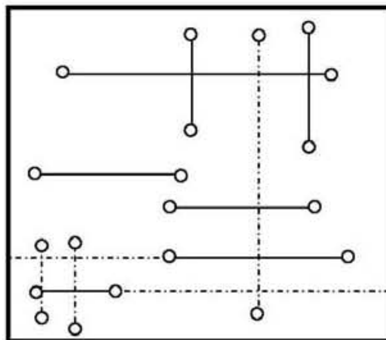


Figura 4.3 Estado del algoritmo de intersección de barrido del plano

4.4.4 Divide y vencerás

El término Divide y Vencerás en su acepción más amplia es algo más que una técnica de diseño de algoritmos. De hecho, suele ser considerada una filosofía general para resolver problemas y de aquí que su nombre no sólo forme parte del vocabulario informático, sino que también se utiliza en muchos otros ámbitos. [172]

Es una técnica de diseño de algoritmos que consiste en resolver un problema a partir de la solución de sub problemas del mismo tipo, pero de menor tamaño. Si los sub problemas son todavía relativamente grandes se aplicará de nuevo esta técnica hasta alcanzar sub problemas lo suficientemente pequeños para ser solucionados directamente. Ello naturalmente sugiere el uso de la recursión en las implementaciones de estos algoritmos.

La resolución de un problema mediante esta técnica consta fundamentalmente de los siguientes pasos:

1) En primer lugar ha de plantearse el problema de forma que pueda ser descompuesto en *k* subproblemas del mismo tipo, pero de menor tamaño. Es decir, si el tamaño de la entrada es n, hemos de conseguir dividir el problema en k subproblemas (donde $1 \leq k \leq n$), cada uno con una entrada de tamaño n_k y donde $0 \leq n_k < n$. A esta tarea se le conoce como división.
2) En segundo lugar han de resolverse independientemente todos los subproblemas, bien directamente si son elementales o bien de forma recursiva. El hecho de que el tamaño de los subproblemas sea estrictamente menor que el tamaño original del problema nos garantiza la convergencia hacia los casos elementales, también denominados casos base.
3) Por último, combinar las soluciones obtenidas en el paso anterior para construir la solución del problema original.

Hay ciertas consideraciones que deben tenerse en cuenta, en primer lugar, el número *k* debe ser pequeño e independiente de una entrada determinada. En el caso particular de los algoritmos divide y vencerás que contienen sólo una llamada recursiva, es decir $k = 1$, hablaremos de algoritmos de *simplificación*. Tal es el caso del algoritmo recursivo que resuelve el cálculo del factorial de un número, que sencillamente reduce el problema a otro subproblema del mismo tipo de tamaño más pequeño. También son algoritmos de simplificación el de búsqueda binaria en un vector o el que resuelve el problema del *k*-ésimo elemento.

La ventaja de los algoritmos de simplificación es que consiguen reducir el tamaño del problema en cada paso, por lo que sus tiempos de ejecución suelen ser muy buenos (normalmente de orden logarítmico o lineal). Además pueden admitir una mejora adicional,

puesto que en ellos suele poder eliminarse fácilmente la recursión mediante el uso de un bucle iterativo, lo que conlleva menores tiempos de ejecución y menor complejidad espacial al no utilizar la pila de recursión, aunque en contra, también en detrimento de la legibilidad del código resultante.

Por el hecho de usar un diseño recursivo, los algoritmos diseñados mediante la técnica de Divide y Vencerás van a heredar las ventajas e inconvenientes que la recursión plantea:

a) Por un lado el diseño que se obtiene suele ser simple, claro, robusto y elegante, lo que da lugar a una mayor legibilidad y facilidad de depuración y mantenimiento del código obtenido.
b) Sin embargo, los diseños recursivos conllevan normalmente un mayor tiempo de ejecución que los iterativos, además de la complejidad espacial que puede representar el uso de la pila de recursión.

Desde un punto de vista de la eficiencia de los algoritmos Divide y Vencerás, es muy importante conseguir que los subproblemas sean independientes, es decir, que no exista solapamiento entre ellos. De lo contrario el tiempo de ejecución de estos algoritmos será exponencial. Como ejemplo pensemos en el cálculo de la sucesión de Fibonacci, el cual, a pesar de ajustarse al esquema general y de tener sólo dos llamadas recursivas, tan sólo se puede considerar un algoritmo recursivo pero no clasificarlo como diseño Divide y Vencerás. Esta técnica está concebida para resolver problemas de manera eficiente y evidentemente este algoritmo, con tiempo de ejecución exponencial, no lo es.

En cuanto a la eficiencia hay que tener en también en consideración un factor importante durante el diseño del algoritmo: el número de subproblemas y su tamaño, pues esto influye de forma notable en la complejidad del algoritmo resultante. Para un estudio más detallado respecto a este paradigma se sugiere el análisis realizado por [172].

En definitiva, el diseño Divide y Vencerás produce algoritmos recursivos cuyo tiempo de ejecución se puede expresar mediante una ecuación en recurrencia del tipo:

$$T(n)=\begin{cases} cn^k & si\ 1\leq n\leq b \\ aT(n\ /\ b)+cn^k & si\ n\geq b \end{cases}$$

donde a, c y k son números reales, n y b son números naturales, y donde $a>0$, $c>0$, $k\geq0$ y $b>1$. El valor de a representa el número de subproblemas, n/b es el tamaño de cada uno de ellos, y la expresión cnk representa el coste de descomponer el problema inicial en los a subproblemas y el de combinar las soluciones para producir la solución del problema original, o bien el de resolver un problema elemental. La solución a esta ecuación, puede

alcanzar distintas complejidades. Recordemos que el orden de complejidad de la solución a esta ecuación es:

$$T(n) = \begin{cases} \Theta(n^k) & si\ a \prec b^k \\ \Theta(n^k \ \log n) & si\ a = b^k \\ \Theta(n^{\log_b a}) & si\ a \succ b^k \end{cases}$$

Las diferencias surgen de los distintos valores que pueden tomar *a* y *b*, que en definitiva determinan el número de subproblemas y su tamaño. Lo importante es observar que en todos los casos la complejidad es de orden polinómico o polilogarítmico pero nunca exponencial, aunque existen algoritmos recursivos que pueden alcanzar esta complejidad en muchos casos. Esto se debe normalmente a la repetición de los cálculos que se produce al existir solapamiento en los subproblemas en los que se descompone el problema original.

Para aquellos problemas en los que la solución haya de construirse a partir de las soluciones de subproblemas entre los que se produzca necesariamente solapamiento existe otra técnica de diseño más apropiada, y que permite eliminar el problema de la complejidad exponencial debida a la repetición de cálculos.

4.4.5 Ramificación y Acotamiento

Esta técnica de diseño o paradigma, cuyo nombre en castellano proviene del término inglés *Branch and Bound (BB)*, se aplica normalmente para resolver problemas de optimización. Ramificación y Poda (RP), al igual que el diseño Vuelta Atrás, realiza una enumeración parcial del espacio de soluciones basándose en la generación de un árbol de expansión.

El diseño RP en su versión más sencilla puede seguir un recorrido en anchura (estrategia LIFO) o en profundidad (estrategia FIFO), o utilizando el cálculo de funciones de costo para seleccionar el nodo que en principio parece más prometedor (estrategia de mínimo costo o LC).

Además de estas estrategias, la técnica de RP utiliza cotas para podar aquéllas ramas del árbol que no conducen a la solución óptima. Para ello calcula en cada nodo una cota del posible valor de aquellas soluciones alcanzables desde ése. Si la cota muestra que cualquiera de estas soluciones tiene que ser necesariamente peor que la mejor solución hallada hasta el momento no necesitamos seguir explorando por esa rama del árbol, lo que permite realizar el proceso de poda.

Básicamente, en un algoritmo de Ramificación y Poda se realizan tres etapas. La primera de ellas, denominada de *Selección*, se encarga de extraer un nodo de entre el conjunto de los nodos vivos. La forma de escogerlo va a depender directamente de la estrategia de búsqueda que decidamos para el algoritmo. En la segunda etapa, la *Ramificación*, se construyen los posibles nodos hijos del nodo seleccionado en el paso anterior. Por último se realiza la tercera etapa, la *Poda*, en la que se eliminan algunos de los nodos creados en la etapa anterior. Esto contribuye a disminuir en lo posible el espacio de búsqueda y así atenuar la complejidad de estos algoritmos basados en la exploración de un árbol de posibilidades. Aquellos nodos no podados pasan a formar parte del conjunto de nodos vivos, y se comienza de nuevo por el proceso de selección. El algoritmo finaliza cuando encuentra la solución, o bien cuando se agota el conjunto de nodos vivos.

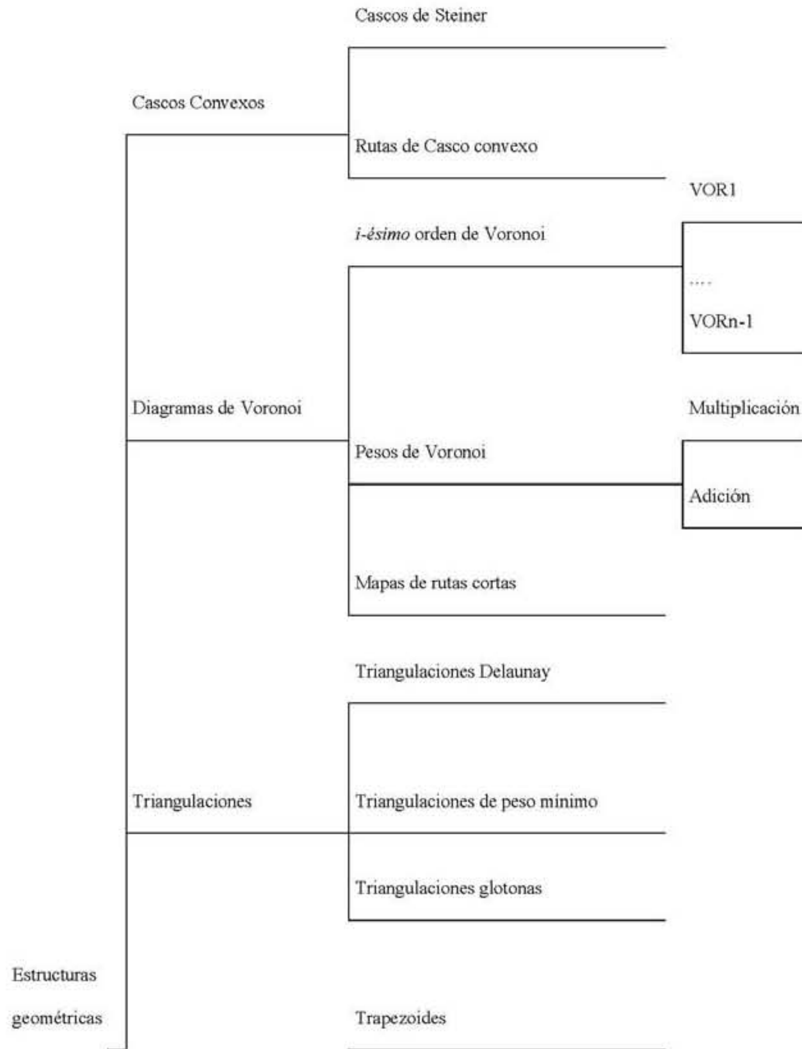
4.5 Estructuras de datos geométricos

En el pasado, los problemas de Topología de Diseño de Redes *TND* por sus siglas en inglés, eran resueltos con estructuras geométricas muy simples (matrices de distancias). Una contribución clave de la *GC* ha sido la introducción de nuevas estructuras geométricas (cascos convexos, diagramas de Voronoi, triangulaciones Delaunay, descomposiciones planares) y el diseño eficiente de algoritmos para su construcción. La construcción de estas estructuras geométricas es compleja. Esto involucra el uso de varios paradigmas de *GC* y estructuras de datos muy elaboradas. Sin embargo, la información y los beneficios computacionales lejos de perjudicar, compensan estas desventajas.

Las estructuras de datos así como los paradigmas utilizados en el análisis de algoritmos juegan un papel importantísimo, ya que, haciendo un estudio exhaustivo de las características del problema a resolver, se pueden insertar la estructura geométrica más adecuada para hacer la representación de un tour como es el caso de este proyecto y también el paradigma más adecuado para lograr la mayor eficiencia en los algoritmos propuestos o utilizados. Bentley y Friedman 1975, [177], hacen un estudio para determinar la representación más adecuada de un tour mediante un árbol K-*dimensional*.

Un árbol K-*dimensional* [WWW5] (k-d) es un árbol de decisión que sirve para hallar una respuesta o curso de acción, dadas k variables o entradas. Cada nodo suyo hace una pregunta sobre una variable; para k variables, puede haber más de k preguntas antes de emitir una respuesta. Las hojas (nodos finales) no hacen pregunta, sino que "dan la respuesta" o curso a tomar. Tradicionalmente [177] los árboles k-d han sido utilizados para hallar la información asociada a ciertos valores de k variables. Por ejemplo, ¿Qué información tenemos para tipo=avión, destino=Habana, fecha=miércoles? La respuesta puede ser: el vuelo Cubana 465. A veces la respuesta es f (no hay, no sé). Hace el papel, aunque no utiliza las técnicas, de un archivo cuya llave es la concatenación de las k

variables. Se muestra en la figura 4.4 una lista que abarca la mayoría de estructuras geométricas que se utilizan de manera muy frecuente en la GC.



de *GC*		
		Triángulos
	Subdivisiones planares	Cadenas monotonas
		Gráficas aciclicas dirigidas
		Búsqueda de árboles multidimensionales
		Arboles de rango
	Descomposiciones celulares	Arboles poligono
		Descomposiciones de caja
		Descomposiciones de cono
		Intervalos de arboles
	Otros	Busqueda en arboles
		Quadtrees
		Octant tree

Figura 4.4. Estructuras de datos geométricos

A continuación se hace la descripción de algunos de los algoritmos que calculan cascos convexos , esta estructura se utilizó para hacer más eficientes a las técnicas heurísticas de inserción, además se hace la descripción del algoritmo de identificación de cruces, otra estructura geométrica importante para poder identificar el espacio de soluciones de la técnica 2-opt.

4.5.1 Cascos Convexos *CH(S)*

El casco convexo o convex hull es uno de los constructores geométricos más fundamentales.

Una idea intuitiva del significado del casco convexo es el contenido de la figura que formaría una banda elástica que rodeara a una nube de puntos una vez que la soltáramos.

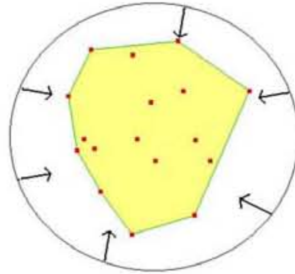


Figura 4.5 Casco convexo

El problema de computar un convex hull no sólo está centrado en aplicaciones prácticas, sino también es un vehículo para la solución de un número de cuestiones aparentemente sin relación con él, que surgen en la Geometría Computacional.

El cálculo del casco convexo de una nube finita de puntos, especialmente en el plano, ha sido exhaustivamente estudiado y tiene aplicaciones, como por ejemplo, en el procesado de imágenes y en localización.

4.5.1.1 Definiciones

Definición: Un conjunto S se dice convexo si dado cualquier par de puntos x, y pertenecientes al conjunto se cumple que el segmento xy que forman está contenido en dicho conjunto S.

Lema: La intersección formada por cualquier grupo de conjuntos convexos dará como resultado un único conjunto que además también será convexo.

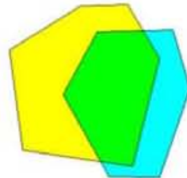


Figura 4.6 Intersección de polígonos convexos

En cambio, como puede verse en la siguiente imagen, la intersección de conjuntos no convexos no tiene por que dar como resultado un conjunto único.

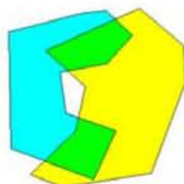


Figura 4.7 Intersección de polígonos no convexos

Definición: Dado un conjunto de puntos finito *S*, se dice que *CH(S)* es la envolvente o casco convexo de *S* si:

a. CH(S) es convexo.

b. S está contenido en CH(S).

c. CH(C) es el menor convexo que contiene a S.

Esta definición se puede extender para conjuntos infinitos, aunque no es el caso que nos interesa.

Los puntos de la frontera del casco convexo se llaman vértices o extremos del casco convexo, pero también se les suele denominar, con el mismo nombre de casco convexo.

El casco convexo de una nube de puntos es siempre un polígono.

4.5.2 Algoritmos para calcular cascos convexos

Hay muchos métodos para el cálculo del casco convexo, ya sea para dos, tres o más dimensiones. En este proyecto se analizará sólo algunos algoritmos para averiguar el casco convexo de puntos situados sobre dos dimensiones.

Teorema 1: Es posible calcular el casco convexo de un convexo *S* en tiempo *O(n log n)*.

Se puede calcular el casco convexo de una nube de puntos a partir de su diagrama de Voronoi en tiempo lineal, lo que junto al tiempo requerido para el cálculo del diagrama de Voronoi nos da un costo de *O(n log n)*, que es el tiempo óptimo. Sin embargo, las constantes multiplicativas son tan grandes que el método no es práctico si no tenemos calculado de antemano el diagrama de Voronoi.

4.5.2.1 Algoritmo Quick Hull

La idea del Quick Hull es ir descartando lo más pronto posible los puntos que no formarán parte de la frontera del casco convexo, que suelen ser los más interiores de la nube de puntos. A continuación se describe el algoritmo.

Paso 1: Encontrar los cuatro puntos extremos de la nube de puntos (norte, sur, este y oeste) y formar un cuadrilátero con estos puntos como esquinas (aunque podría darse el caso de obtenerse sólo un triángulo e incluso una línea, lo cual no afecta el funcionamiento del algoritmo). Todos los puntos que hayan quedado dentro de este cuadrilátero no formarán ya parte de la frontera.

Paso 2: Los puntos exteriores al cuadrilátero se encontrarán divididos en cuatro (o menos) zonas no comunicadas.

En cada una de estas regiones obtendremos el punto más alejado al lado del cuadrilátero adyacente a dicha zona. De esta forma obtendremos una figura de hasta ocho vértices que descartará los puntos interiores como pertenecientes al borde del casco convexo y dividirá los puntos exteriores en hasta ocho nuevas regiones.

Paso 3: Seguiremos actuando para cada nueva región según las reglas anteriores hasta que no queden vértices externos a la figura. Esta figura resultante será el casco convexo. En Figura 4.7 se puede observar la ejecución del algoritmo Quick Hull paso a paso.

Teorema 2: El algoritmo Quick Hull calcula el casco convexo en tiempo $O(n^2)$.

A pesar de ser de orden cuadrático, suele tardar menos tiempo que calculando el casco convexo a través del diagrama de Voronoi.

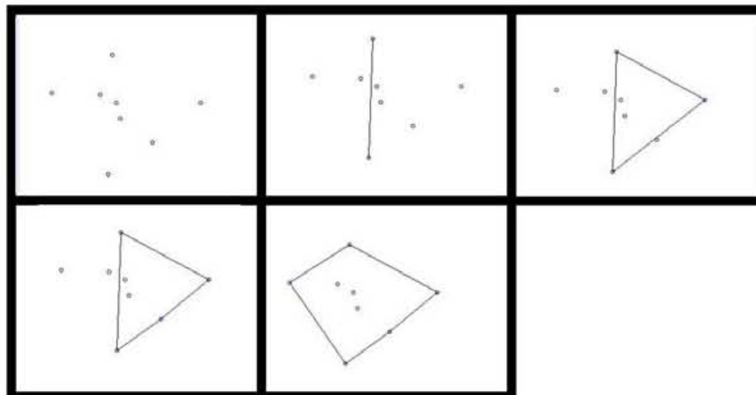


Figura 4.8 Ejecución del algoritmo Quik Hull

4.5.2.2 Algoritmo Scan de Graham

Este algoritmo utiliza una lista en la que va almacenando y ordenando correctamente los puntos que constituyen los extremos del casco convexo.

Paso 1: Elegimos un punto, que será el que menor coordenada *y* posea (así nos aseguramos que pertenezca al casco convexo). Este punto será nuestro punto inicial. A continuación vamos almacenando el resto en la lista, ordenados según el valor del ángulo que forman con el primer punto que escogimos.

Paso 2: Recorremos la lista que hemos formado, tomando en cada momento tres puntos: Inicial, Medio y Final.

Comprobamos si el ángulo que forman Inicial, Medio y Final (IMF) es negativo (sentido horario) o positivo (sentido antihorario).

Si es positivo, Medio pasará a ser Inicial, Final pasará a ser Medio y el siguiente elemento de la lista pasará a ser Final.

Si es negativo, Medio será borrado de la lista, Inicial pasará a ser Medio, el anterior elemento de la lista pasará a ser Inicial, y Final queda inalterado.

La lista obtenida al finalizar con todos los elementos de la lista será el casco convexo de la nube de puntos.

Teorema 3: El algoritmo Scan de Graham calcula el casco convexo en tiempo $O(n \log n)$.

4.5.2.3 Algoritmo incremental

La idea básica es ir añadiendo puntos mientras vamos modificando el casco convexo.

Paso 1: Elegimos un punto. Si el nuevo punto está dentro del casco, no hay nada que hacer. En otro caso, borramos todos los bordes que el polígono pueda ver, es decir, que trazando una línea desde el punto no choquemos con ninguna otra.

Paso 2: Añadimos dos líneas para conectar el nuevo punto al resto del antiguo casco.

Paso 3: Repetimos de nuevo desde el paso uno para los puntos que estén fuera del convex hull actual, hasta que todos los vértices estén dentro.

Si usamos una estructura para el casco que nos permita tomar una línea adyacente a otra en tiempo constante, podremos encontrar un lado visible en tiempo lineal y todos los demás en tiempo constante por lado. La actualización toma tiempo lineal. El tiempo total es de $O(n^2)$.

Teorema 4: El algoritmo incremental calcula el casco convexo en tiempo $O(n^2)$.

4.5.2.4 Algoritmo divide y vencerás

Paso 1: Ordenamos los puntos según la coordenada *x*.

Paso 2: Dividimos los puntos en dos bloques, izquierda y derecha (L y R), de igual número de elementos, que cumplen que el punto más a la derecha de L está más a la izquierda que el punto más a la izquierda de R.

Paso 3: Recursivamente, encontramos el convex hull de *L* y *R*. Para mezclar el cierre de *L* y el de *R* es necesario unirlos utilizando las tangentes comunes más altas y bajas. La tangente superior común pude descubrirse en tiempo lineal explorando alrededor del cierre de *L* en el sentido de las agujas del reloj y en el cierre de *R* en el sentido antihorario. Para la tangente inferior giraremos en los sentidos inversos.

Las líneas que queden dentro del casco que formamos al unir los cierres de *L* y *R* serán borradas.

Debido a que la mezcla no puede realizarse en tiempo lineal el casco convexo puede ser encontrado en tiempo $O(n \log n)$.

Teorema 5: El algoritmo divide y vencerás calcula el casco convexo en tiempo $O(n \log n)$.

En este proyecto en especial se utilizo el algoritmo de envoltura de regalo (Gift Wrapping) para calcular el casco convexo del conjunto de instancias de la TSPLIB.

4.5.2.5 Algoritmo envoltura de regalo (Gift Wrapping)

Para este caso en especial de dos dimensiones, este algoritmo también es conocido como el algoritmo de la marcha de Jarvis, ya que es quien lo propone en 1973. La descripción del algoritmo asume que los puntos están colocados en posición general, por ejemplo que no son colineales, sin embargo el algoritmo se puede modificar para tratar con colinearidad e incluso si el casco convexo tuviera uno o dos vértices. También se puede extender a dimeniones mayores.

Paso 1: Elegimos el punto A, que será el que menor coordenada *y* posea (así nos aseguramos que pertenezca al casco convexo). Este punto será nuestro punto inicial.

Paso 2: Puede encontrarse otro punto B, que cumplirá que todos los puntos están situados a la izquierda de AB.

Paso 3: Similarmente, podemos encontrar C allí donde todos los puntos estén a la izquierda de BC. Repetimos este método para el resto de puntos.

El algoritmo comienza con i=0 y un punto p_0 que se sabe pertenece al casco convexo, por ejemplo el punto extremo izquierdo y seleccionar el punto p_{i+1} de tal manera que todos los puntos estén a la derecha de la línea p_i p_{i+1}. Este punto se puede encontrar en un tiempo *O(n)* comparando ángulos polares de todos los puntos con respecto al punto p_i. Se deja que i=i+1, y el proceso se repite hasta que p_h=p_0.

La complejidad de este algoritmo será de *O(nh)*, donde *n* es el número de puntos y *h* es el número de vértices del casco convexo. El desempeño en la vida real es favorable cuando *n* es pequeño o bien cuando *h* se espera ser muy pequeño con respecto a *n*.

Teorema 6: El algoritmo envoltura de regalo calcula el casco convexo en tiempo *O(nh)*.

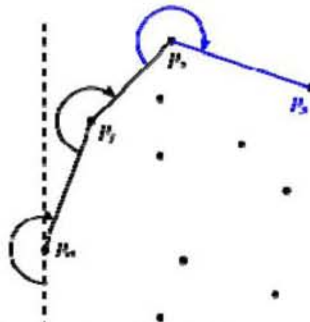


Figura 4.9 Ejecución del algoritmo envoltura de regalo

A continuación se muestra el pseudocódigo del algoritmo

```
jarvis(S)
   pointOnHull = leftmost point in S
   i = 0
   repeat
      P[i] = pointOnHull
      endpoint = S[0] //initial endpoint for a candidate edge on the hull
      if (endpoint == P[i])
         endpoint = S[1]       // endpoint should not be equal to P[i]
      if (S[j] is on left of line from P[i] to endpoint)
           endpoint = S[j]     // found greater left turn, update endpoint
      i = i+1
      pointOnHull = endpoint
   until endpoint == P[0]       // wrapped around to first hull point
```

Este algoritmo contiene dos loops, uno revisa todos los puntos de la instancia y otro loop revisa los puntos que pertenecen al casco convexo. Por lo tanto el tiempo de corrida es *nh*. Este tiempo depende del tamaño de los datos de entrada y los de salida, lo que se conoce como un algoritmo sensible a los datos de salida. Para ver los detalles de los algoritmos que calculan el casco convexo en segunda y tercera dimensión se sugiere la siguiente página [WWW6].

Otra de las estructuras geométricas que se utilizan en este proyecto es la intersección de segmentos de línea.

4.6 Intersección de segmentos de línea

En este problema se proporciona una lista de segmentos de línea y se determina si cualquier par de estos se intersecan. Un algoritmo ingenuo examina cada par de segmentos, pero para un número elevado de intersección de segmentos posiblemente esto se vuelve cada vez más ineficiente ya que la búsqueda se tiene que hacer por cada segmento de línea y en el peor de los casos se tiene un tiempo de $O(n^2)$. Una forma común y más eficiente de resolver este problema para un gran número de segmentos es utilizar un algoritmo de línea de barrido, es un tipo de algoritmo que usa una línea o superficie de barrido para resolver varios problemas en un espacio euclidiano. Es una técnica clave en geometria computacional.

La idea central detrás de este algoritmo es imaginar que una línea (la mayoría de las veces vertical) se barre o se mueve a través del plano empleando operaciones geométricas para detectar las intersecciones y la solución completa está disponible una vez que la línea ha pasado por todos los objetos en este caso en todos los segmentos de línea.

Se utiliza una estructura de datos dinámica basada en árboles binarios. El algoritmo Shamos-Hoey aplica este principio para detectar la intersección de segmentos de línea, también el algoritmo de Bentley-Ottmann actúa bajo el mismo principio. [178] Sin embargo, este algoritmo no logra las cotas inferiores teóricas y se le reconoce como unos de los primeros algoritmos "sensibles a los datos de salida" en inglés "output-sensitive algorithm".

Los algoritmos sensibles a los datos de salida se identifican porque su eficiencia depende tanto de los datos de entrada como de los datos de salida. Aquí los datos de entrada es un conjunto Ω n segmentos, y los datos de salida es un conjunto Λ de k intersecciones calculadas. En los primeros algoritmos como el propuesto por Shamos y Hoey en 1976, [179] muestran como detectar cuando existe al menos una intersección en un tiempo $O(n \log n)$ en un espacio $O(n)$ barriendo sobre un orden lineal de Ω Bentley y Ottmann en 1979 [176] extienden esta idea y proponen su algoritmo que calcula todas las intersecciones k en $O((n+k) \log n)$ en un espacio $O(n+k)$.

Para efectos de este trabajo se implementó el algoritmo de Fuerza Bruta y el de Bentley-Ottman para identificar todas las aristas que se intersecan.

Los datos de entrada serán:

Input: $\Omega=\{s1, s2, \ldots, sn\}$ de n segmentos en el plano.

Output: conjunto Λ de puntos de intersección entre los segmentos en S. (Con los segmentos que contienen cada punto de intersección).

A continuación se describen a detalle los dos algoritmos para la identificación de las aristas que se cruzan. Estos son fuerza bruta y algoritmo de barrido de plano Bentley-Ottman.

4.6.1 Algoritmo de Fuerza Bruta

En general para un conjunto de n segmentos hay hasta $O(n^2)$ puntos de intersección, ya que si todos los segmentos se intersecan unos con otros habría $(n-1)+(n-2)+\ldots+1=n(n-1)/2=O(n^2)$ puntos de intersección. En el peor de los casos calcularlos todos llevaría $O(n^2)$

Este algoritmo toma todos los pares de aristas y se pregunta si se existe intersección entre ellas. Esto significa mucho tiempo de cálculo, sin embargo, cuando hay pocos puntos de intersección o sólo cuando se necesita detectar únicamente un punto, se tiene un algoritmo eficiente. Cabe señalar que en este proyecto no interesa encontrar el punto de intersección sino únicamente las aristas que se intersecan.

4.6.2 Algoritmo de Barrido de Plano

Básicamente este algoritmo consiste en efectuar un desplazamiento de una recta vertical imaginaria en el plano deteniéndose en los extremos de los segmentos y en los puntos de intersección detectados. Si observamos la situación, dicha recta vertical o línea barredora corta a los segmentos dados o no dependiendo de su posición.

Los segmentos a los que corta dicha recta, así como la posición relativa de los puntos de corte en la línea barredora, sólo varían cuando la recta pasa por uno de los puntos mencionados: extremo inicial de un segmento (en este caso un nuevo segmento entra en acción) , extremo final de un segmento (en este caso un segmento desaparece de la recta barredora) y en los puntos de intersección de dos segmentos (en los cuales cambia la posición relativa de los puntos de intersección con la recta barredora).

Por otra parte, para que dos segmentos lleguen a cortarse, es preciso que en algún momento ambos segmentos corten a la recta barredora y sean consecutivos dentro de la misma sus puntos de intersección con ella.

Gracias a las observaciones anteriores, para hallar todos los puntos de intersección, no es preciso comparar todos los segmentos con todos, sino que basta que estudiemos si se cortan aquellos pares de segmentos que sean consecutivos en su corte con la recta barredora en algún instante.

Específicamente los datos de entrada del algoritmo es una colección de $\mathbf{\Omega} = \{\mathbf{L}_i\}$ segmentos de línea $\mathbf{L}_i$, y los datos de salida serán un conjunto $\mathbf{\Lambda} = \{I_j\}$ de puntos de intersección. Este algoritmo también se le conoce como "*algoritmo de línea de barrido*" porque la operación básica de puede ser visualizada como teniendo una línea (LB o **SL** por sus siglas en inglés), barriendo sobre la colección $\mathbf{\Omega}$ y recolectando información a medida que pasa sobre cada segmento $\mathbf{L}_i$. La información recolectada es una lista ordenada de todos los segmentos en $\mathbf{\Omega}$ que se encuentran intersecados por la LB. La estructura de dato que mantiene esta información se llama "línea de barrido" y el proceso por el cual descubre las intersecciones es la eficiencia y el corazón del algoritmo.

Para implementar la lógica de barrido, se debe ordenar de manera lineal los segmentos de $\mathbf{\Omega}$ para determinar el orden por el cual la LB las encontrará. De hecho, se necesitan ordenar los puntos extremos $\{E_{i0}, E_{i1}\}$ de todos los segmentos $\mathbf{L}_i$ así que se puede detectar cuando la LB cruza a todos estos segmentos. Los puntos extremos están ordenados por los valores de x de manera creciente y después de la misma manera la coordenada y. De manera tradicional, la línea de barrido es vertical y se traslada de derecha a izquierda como se muestra en el siguiente figura 4.10.

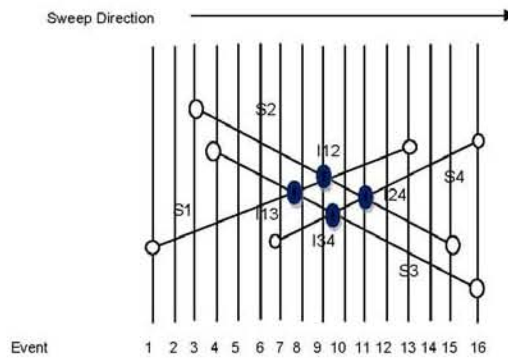


Figura 4.10 movimiento de la línea de barrido

En cualquier punto en el algoritmo, la línea de barrido SL cruza los segmentos con un punto final a la izquierda de éste y el otro extremo a la derecha. La estructura de datos SL mantiene un seguimiento de estos segmentos por: (1) agregando un segmento cuando su extremo izquierdo se encuentra, y (2) eliminando un segmento cuando su extremo derecho se encuentra. La LB también mantiene los segmentos en una lista ordenada por una relación "arriba-abajo" . Por lo tanto, añadir o eliminar un segmento, su posición en la lista debe ser determinado, y esto se puede realizar en el peor de los casos en un tiempo O (log n) de búsqueda binaria de los segmentos que se encuentra en la LB. Pero, además de añadir o eliminar segmentos, hay otro acontecimiento que cambia la estructura de la LB, a saber, cuando se cruzan dos segmentos, después, sus posiciones en la lista ordenada se intercambian. Teniendo en cuenta los dos segmentos, los cuales deben ser vecinos en la lista, este intercambio es una operación O (log n).

Para organizar todo esto, el algoritmo tiene un "evento cola", "***event queue***" ξ cuyos elementos causan un cambio en la lista de segmentos de la LB. Inicialmente, ξ es una lista de segmentos de la LB ordenada por los extremos del segmento. Y ya que las intersecciones se encuentran, también se agregan a ξ en el mismo orden que el utilizado en los puntos extremos. Uno debe probar para evitar insertar doblemente las intersecciones en el ***event queue***. En la figura 4.10 se muestra cómo puede suceder esto. En el evento 2, los segmentos s1 y s2 causan una intersección I12 para ser calculada y puesta en el evento cola. En el evento 3 el segmento s3 se presenta y divide a los segmentos s1 y s2. Después en el evento 4, s1 y s3 intercambian lugares sobre la línea de barrido, y s1 se pone junto a s2 provocando de nuevo el cálculo de la intersección I12. Pero solamente puede haber un evento por cada intersección e I12 no puede ser puesta en el evento cola nuevamente.

Así, cuando una intersección es puesta en el evento cola, se tiene que identificar si ya está dentro del evento. Ya que hay a lo más un punto de intersección por cada dos segmentos, etiquetar la intersección será suficiente para su pronta identificación. Como resultado de todo esto, el tamaño máximo de un evento cola es igual a $2n + k \leq 2n + n^2$ y una inserción dentro de este evento en el peor de los casos puede lograrse en un tiempo $O(\log (2n + n^2)) = O(\log n)$.

Este algoritmo es eficiente en encontrar el conjunto de segmentos que se intersecan, ya que los segmentos son actualizados secuencialmente sobre la lista de la LB, y se puede determinar si existe una posible intersección. Cuando se encuentra una intersección válida entonces se inserta en el evento cola y más aún, cuando una intersección sobre ξ se procesa mediante el barrido, entonces se reordena la lista de la LB y se agrega a la lista de salida Λ. Al final cuando todos los eventos han sido procesados, Λ contendrá el conjunto completo de todas las intersecciones.

El cálculo de una intersección solo se efectúa si los dos segmentos vecinos tienen un orden de arriba-abajo sobre la línea de barrido. Por lo tanto, sólo hay unos casos restringidos para los cuales se realicen los cálculos de intersección de segmentos:

1. Cuando un segmento se agrega a la LB, determinar si se cruza con sus vecinos de arriba y abajo.
2. Cuando un segmento es eliminado de la LB, sus vecinos previos de arriba y abajo traen a nuevos vecinos. Por lo tanto se necesita saber si existe una intersección entre estos.
3. En una intersección de un evento cola, dos segmentos intercambian sus posiciones en la LB, y su intersección con sus nuevos vecinos deben ser determinados.

Esto significa que para el procesamiento de cualquier evento (punto extremo o intersección) de ξ se requieren calcular a lo más 2 intersecciones. Para determinar el tiempo necesario que se requiere para agregar, encontrar, intercambiar y remover segmentos de la estructura de dato de la LB es necesario implementarla como un árbol binario balanceado (como un árbol negro-rojo) que garantiza que estas operaciones tomen a lo mas $O(\log n)$ ya que n es el tamaño máximo de la LB. Para cada $(2n+k)$ eventos tienen en el peor de los casos se procesan en un tiempo $O(\log n)$. El tiempo total de ejecución será: O(orden inicial)+O(procesamiento del evento)= $O(n \log n) + O((2n+k) \log n) = O((n+k) \log n)$.

4.6.2.1 Pseudocódigo del algoritmo Bentley-Ottman

La implementación de este algoritmo viene dado por el siguiente pseudocódigo:

```
Initialize event queue ξ = all segment endpoints;
    Sort ξ by increasing x and y;
    Initialize sweep line SL to be empty;
    Initialize output intersection list □ to be empty;

    While (ξ is nonempty) {
        Let E = the next event from ξ;
        If (E is a left endpoint) {
            Let segE = E's segment;
            Add segE to SL;
            Let segA = the segment above segE in SL;
            Let segB = the segment below segE in SL;
            If (I = Intersect( segE with segA) exists)
                Insert I into ξ;
            If (I = Intersect( segE with segB) exists)
                Insert I into ξ;
        }
        Else If (E is a right endpoint) {
            Let segE = E's segment;
            Let segA = the segment above segE in SL;
            Let segB = the segment below segE in SL;
            Remove segE from SL;
            If (I = Intersect( segA with segB) exists)
                If (I is not in ξ already) Insert I into ξ;
        }
        Else {  // E is an intersection event
            Add E to the output list Λ;
            Let segE1 above segE2 be E's intersecting segments in SL;
            Swap their positions so that segE2 is now above segE1;
            Let segA = the segment above segE2 in SL;
            Let segB = the segment below segE1 in SL;
            If (I = Intersect(segE2 with segA) exists)
                If (I is not in ξ already) Insert I into ξ;
            If (I = Intersect(segE1 with segB) exists)
                If (I is not in ξ already) Insert I into ξ;
        }
        remove E from ξ;
    }
  return ξ;
}
```

Esta rutina calcula la lista completa de todos los puntos de intersección.

Sin embargo, si únicamente queremos saber si existe una intersección la rutina puede terminar tan pronto se detecte una.

Por lo tanto, este algoritmo sólo necesita un espacio de O(*n*) y corre en un tiempo O(*n* log *n*). Este es el algoritmo original de [Shamos y Hoey, 1976]. El pseudo-código para esta rutina simplificado es:

```
Initialize event queue ξ = all segment endpoints;
Sort ξ by increasing x and y;
Initialize sweep line SL to be empty;

While (ξ is nonempty) {
    Let E = the next event from ξ;
    If (E is a left endpoint) {
        Let segE = E's segment;
        Add segE to SL;
        Let segA = the segment above segE in SL;
        Let segB = the segment below segE in SL;
        If (I = Intersect( segE with segA) exists)
            return TRUE;   // an Intersect Exists
        If (I = Intersect( segE with segB) exists)
            return TRUE;   // an Intersect Exists
    }
    Else {  // E is a right endpoint
        Let segE = E's segment;
        Let segA = the segment above segE in SL;
        Let segB = the segment below segE in SL;
        Delete segE from SL;
        If (I = Intersect( segA with segB) exists)
            return TRUE;   // an Intersect Exists
    }
    remove E from ξ;
}
return FALSE;       // No Intersections
```

4.6.2.2 Algoritmo Intersección de segmentos con barrido de plano O(n *log* n)

A continuación se puede observar este algoritmo mediante una serie de pasos explicado con palabras.

Entrada: n segmentos del plano dados por las coordenadas de sus extremos.

Paso 1: Ordenar con los valores de las coordenadas x y y de manera creciente de los 2n extremos de los segmentos en una lista E y considerar tres listas vacías A, L e I.

Paso 2: Si E es una lista vacía el conjunto de los puntos de intersección es I. FIN

Paso 3: p es MIN(E) . Si p es extremo izquierdo de un segmento, entonces insertar s en la lista L, y hallar los elementos s1, s2, anterior y posterior a s en L, respectivamente. Si s1

corta a s entonces añadir el par (s1,s2) a la lista A. Si s2 corta a s entonces añadir el par (s2,s) a la lista A.

Paso 4: Si p es extremo derecho de un segmento s, entonces hallar los elementos anterior y posterior a s en s1 y s2. Borrar s de la lista L. Si s1 corta a s2 añadir el par (s1, s2) a A.

Paso 5: Si p es n punto de intersección de dos segmentos s1 y s2 (siendo s1 anterior a s2 en la lista L a la izquierda del punto p), hallar el elemento s3 de L anterior a s1 en L y el elemento s4 posterior a s2 en L. Intercambiar s1 y s2 en la lista L. Si s3 corta a s2 entonces añadir el par (s3,s2) a la lista A. Si s1 corta a s4 entonces añadir el par (s1,s4) a la lista A.

Paso 6: Si la lista A es no vacía, para cada par (s, s´) de A, sea x el punto de intersección de los segmentos s y s´. Borrar (s, s´) en la lista A. Si x no está en la lista E entonces insertar x en la lista E y añadir x a la lista I.

Paso 7: Volver al paso 2.

Salida: La lista I.

Después de más de 30 años este algoritmo sigue siendo el más popular y el más utilizado en la práctica, ya que es relativamente fácil de entender e implementar. Sin embargo es necesario notar que cuando k es muy grande de orden $O(n^2)$, el algoritmo Bentley-Ottman toma un tiempo de $O(n^2 \log n)$ el cual es peor que el algoritmo de fuerza bruta $O(n^2)$.

Uno puede pensar en implementar el algoritmo de fuerza bruta cuando k se espera que sea más grande que $O(n)$, pero si k se espera ser menor o igual que $O(n)$ se puede utilizar el algoritmo de Bentley-Ottman que se espera tenga en el peor de los casos un tiempo $O(n \log n)$ y en un espacio $O(n)$.

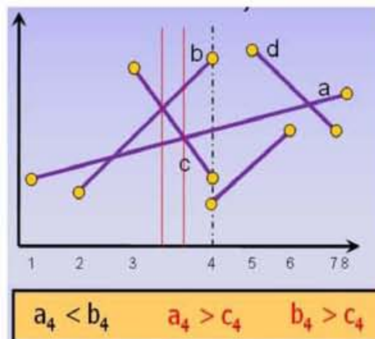


Figura 4.11 Actuación del algoritmo Bentley-Ottman

En el transcurso del siguiente capítulo se observa a través de los estudios experimentales de la metodología propuesta la inclusión de la estructura de casco convexo y el paradigma de barrido de plano para la identificación de los pares de aristas que se intersecan. Se observará que la inclusión de estos elementos de la geometría computacional, permiten una mejor actuación de los algoritmos de construcción y de búsqueda local clásicos.

Esta mejora también es posible ya que el PAVE de manera "*natural*" cumple con la desigualdad del triángulo, lo cual quiere decir que siempre podremos acortar los tours dirigiéndonos de manera directa a una ciudad sin pasar por ciudades intermedias, y el hecho de que si un ciclo Hamiltoniano se cruza a sí mismo, puede ser fácilmente acortado [51], entonces basta con eliminar las dos aristas que se cruzan y reconectar los dos caminos resultantes mediante aristas que no se corten, lo que da como resultado que el ciclo final es más corto que el inicial. Lo cual implica que siempre se puede mejorar un tour utilizando la propiedad "*natural*" del PAVE, en este caso, se logra con la estrategia de la técnica 2-Opt clásica.

Además se observa la ventaja de la inclusión de casco convexo a las técnicas de construcción en una mejora del 2% del circuito final.

Capítulo 5

Metodología propuesta y Experiencia Computacional

5. 1 Condiciones de inicio de la metodología

5.2 Metodología propuesta

5.3 Proceso para la identificación de las aristas que se intersecan

5.3.1 Tipos de cruces en el espacio R2
5.3.2 Identificación del tipo de cruce

5.4 Desarrollo de la metodología propuesta
5.4.1 Detalle de programación

5.5 Resultados
5.5.1 Medición y resultados de la eficiencia en tiempo de los algoritmos de identificación de pares de aristas que se cruzan.
5.5.2 Mejora de las técnicas de inserción con la estructura de casco convexo

5.6 Resumen de resultados de calidad y tiempo de solución con respecto a la solución óptima

El objetivo de este proyecto es lograr mayor eficiencia tanto en los algoritmos de construcción como de búsqueda local que resuelven el problema del agente viajero euclidiano.

Para lograr el objetivo se identificaron las propiedades naturales del problema y se incorporaron elementos de la geometría computacional a las técnicas heurísticas de construcción y de búsqueda local.

Como bien es sabido en el ámbito del diseño y análisis de algoritmos los indicadores más importantes que se toman para determinar si un algoritmo es más eficiente que otro son el tiempo de ejecución y calidad de la solución que devuelven. Diseñar algoritmos que sean eficientes ambos indicadores es una tarea extremadamente difícil de lograr, ya que si se quiere una solución óptima es necesario evaluar un espacio de soluciones mayor lo que conlleva a mayor tiempo de búsqueda. Así que la eficiencia de un parámetro esta en detrimento del otro.

Disminuir el espacio de soluciones conlleva a un menor tiempo de ejecución, e irremediablemente a una solución local más alejada de un óptimo global. Sin embargo la solución que se muestra con la metodología propuesta es alentadora tanto en tiempo como en calidad de la solución. En el caso particular de este proyecto se logra hacer más eficiente en calidad de solución a 6 técnicas de construcción incorporando la estructura geométrica de casco convexo y se mejora el tiempo en la técnica de búsqueda local 2-Opt, incorporando el paradigma de barrido de plano en el algoritmo de identificación de cruces.

La resolución del problema a resolver de este proyecto se llevó a cabo en dos etapas. En la primera los experimentos se llevaron a cabo en el lenguaje de programación Visual Basic 6, sin embargo los resultados no son tan alentadores, ya que los tiempos de ejecución están expresados en minutos e incluso en algunas instancias en horas. Sin embargo, como producto de este trabajo se obtuvo un laboratorio en el cual se pueden observar el comportamiento de las técnicas propuestas cuyas características se pueden observar en el Anexo I.

De esta etapa se puede corroborar que el lenguaje de programación utilizado influye fuertemente en los tiempos de ejecución, siendo necesario un lenguaje que permita hacerlo más eficiente como JAVA.

Como resultado de esta etapa se pudieron obtener resultados preliminares por ejemplo se puede decir que independientemente del criterio con que se tomen los pares de aristas a descruzar el costo del tour final libre de cruces sólo varia en menos de un 1%.

Es importante señalar que el laboratorio propuesto sirve de base para la experimentación y observar paso a paso el comportamiento de la técnica del vecino más cercano, así como las 2-Opt clásica y el algoritmo propuesto, que se detalla en las secciones siguientes. La

interface realizada permite la inclusión de otras técnicas tanto de construcción como de búsqueda local.

En la segunda etapa se migraron todos los algoritmos utilizados al lenguaje de programación JAVA y se incorporaron otras técnicas de construcción, como mejor vecino más cercano y las técnicas de inserción con cuatro criterios diferentes: más barato, más lejano, más cercano y aleatorio, para poder obtener resultados competitivos. Se propuso la siguiente metodología para lograr mayor eficiencia en los algoritmos de construcción como en las de búsqueda local. Antes se explicarán las condiciones del experimento.

5. 1 Condiciones de inicio de la metodología

Instancias

Las instancias que se emplearon para el desarrollo experimental fueron tomadas de la página TSPLIB [WWW1] son un conjunto de 79 en total y varían desde 48 hasta 16,000 ciudades. En este proyecto se tomaron 74 las cuales son menores a 10,000 ciudades.

Equipo de cómputo

Se usaron 6 computadoras Intel Pentium IV, a 3GH y con 1 Gb en RAM.

Algoritmos

Se programaron 6 técnicas de construcción: vecino más cercano, mejor vecino más cercano, inserción más barata, más lejana, más cercana e inserción aleatoria, así como las técnicas 2-opt y la modificación propuesta de esta misma técnica.

Lenguaje de programación

Los algoritmos empleados se programaron en JAVA

5.2 Metodología propuesta

La metodología se puede resumir en dos fases, la fase 1 que es la de construcción de un circuito hamiltoniano y la fase 2 que consta de la mejora de este.

La fase I o fase de construcción está compuesta de dos pasos: el paso 1 consiste de generar la instancia, ya sea de manera aleatoria o bien valerse de instancias previamente construidas. Para el caso especial de este proyecto se emplean el conjunto de 74 instancias de la TSPLIB, sin embargo puede tomarse cualquier instancia, y el paso 2 se construye un circuito Hamiltoniano con cualquier de las seis técnicas de construcción descritas con anterioridad, con y sin la inclusión del elemento de Geometría Computacional de *Casco Convexo*.

La fase de mejora o bien la fase II está compuesta por tres pasos: en el paso 1, se identifican los pares de aristas que se intersecan del circuito hamiltoniano construido previamente y se guardan en una lista L. La identificación de las aristas que se intersecan se lleva a cabo con la estructura de Geometría Computacional *Intersección de segmentos*, con el paradigma de *barrido de plano*, así como *fuerza bruta*. En el paso 2 se toman únicamente las aristas de la lista L bajo tres criterios distintos (FIFO, PRIO y RANDOM) y se descruzan bajo la filosofía de la técnica clásica 2-Opt, es decir se borran y se intercambian estas aristas cuidando de no formar subcircuitos. El algoritmo para cuando la lista L se encuentra vacía, ya que esto indica que no hay más pares de aristas por descruzar. En el paso 3 y último se tiene un circuito libre de cruces. En la figura 5.1 se puede apreciar la metodología propuesta a través de un diagrama de flujo.

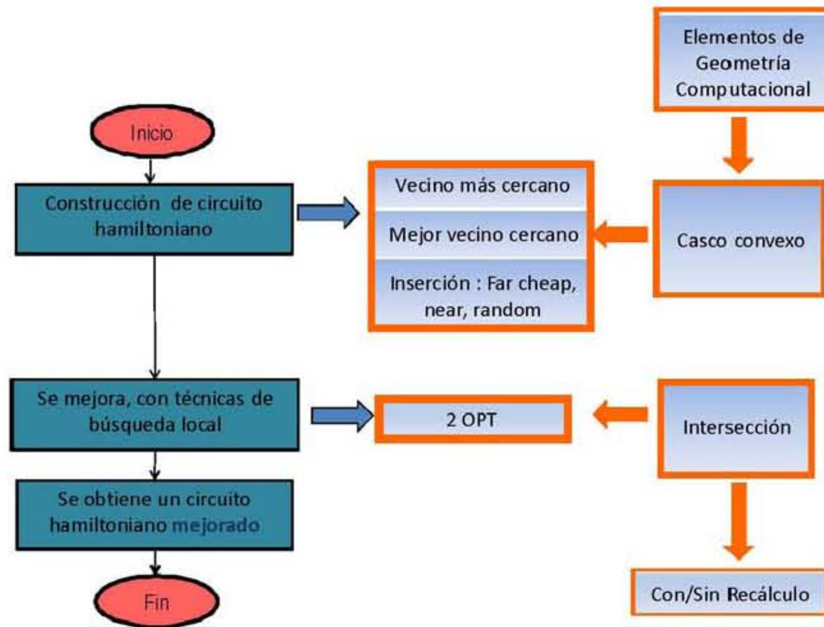


Figura 5.1 Diagrama de flujo de la metodología propuesta.

Descripción detallada de la metodología

Fase I, paso 1: Se construye una instancia de manera aleatoria o bien se manda llamar una instancia pre-construida. Para efectos de este proyecto se utilizó el conjunto de 74 instancias de la TSPLIB menores a 10,000 ciudades.

Fase I, paso 2: Una vez que se tiene la instancias, se construye un circuito hamiltoniano con cualquiera de las 6 técnicas de construcción, vecino más cercano, mejor vecino más cercano, inserción más barata, más lejana, más cercana e inserción aleatoria, cabe señalar que esta parte de la metodología se llevó a cabo con y sin la estructura de casco convexo, para detectar de manera experimental la mejora o no de esta inclusión.

A continuación se muestra el pseudocódigo de este procedimiento, incluyendo la estructura de casco convexo.

```
Paso 1. Se construye una solución inicial o subtour con el algoritmo de
casco convexo (Envoltura de regalo).
Paso 2. (Inserción) Para cada ciudad k no contenida en el subtour, se
decide entre que par de ciudades i y j se inserta la ciudad k. Esto es
para cada ciudad k, encontrar (i,j) tal que, cik+ckj-cij sea el mínimo.
Paso 3. (Selección) De todos los (i,k,j) encontrados en el paso 2,
determinar (i*, k*, j*) tal que (ci*k*+ck*j*)/ci*j* sea el mínimo.
Paso 4. Insertar la ciudad k* en el sobtour entre las ciudades i* y j*.
Paso 5. Repetir los pasos del 2 al 4 hasta obtener un circuito
hamiltoniano o tour.
```

Cabe señalar que el criterio de inserción en el paso 2 tiene varias estrategias, más lejano, más barato, aleatorio y más cercano.

Fase II, paso 1: Una vez construido el circuito se procede a identificar las aristas que se cruzan, se guardan en una lista L. La identificación de los pares de aristas que se intersecan se puede llevar a cabo con dos algoritmos, con fuerza bruta o bien con el de Bentley-Ottman. Este punto es el corazón de la metodología propuesta y se detallará párrafos posteriores.

Fase II, paso 2: Se toman únicamente de esta lista las aristas que se intercambiarán para descruzarlas (son altamente prometedoras), bajo la filosofía de la técnica 2-Opt clásica, k=2, es decir se toman las aristas, se borran y se intercambian cuidando de no formar subcircuitos. Par lograrlo, se representa al circuito euclidiano como una serie sucesiva de puntos $\tau_1 = \{x_1, x_2, x_3, x_4, x_5,\ldots, x_n, x_1\}$ en donde las aristas $\{x_1, x_2\}$, $\{x_2, x_3\}$ hasta $\{x_n, x_1\}$ conforman el circuito. Un movimiento que remueva e intercambia únicamente aquellas aristas que se están intersecando, funciona de la siguiente forma: suponga que los pares de aristas que se están intersecando son: $\{x_1, x_2\}$ y $\{x_3, x_4\}$, apareciendo en este orden en el circuito original. En un sólo movimiento se desconectan y se vuelven a reconectar, intercambiándolas por dos nuevas aristas $\{x_1, x_3\}$ y $\{x_2, x_4\}$, teniendo cuidado que el nuevo circuito siga siendo Hamiltoniano, ya que de no reconectar correctamente el circuito se generan subcircuitos.

Se consideraron tres formas de mandar llamar estas aristas: FIFO, (que consiste en descruzar el primer par de aristas de la lista L), ALEATORIO (que toma cualquier par de aristas) y por último el PRIORIZADO (en el que previamente se identifica de la lista L, la

arista de mayor longitud y es la que se descruza primero). En este proceso se corre el riesgo de formar cruces que de manera inicial no se encontraban, y existen dos posibilidades de calcular la lista L, una puede ser una vez que se descruza el primer par de aristas volver a calcularla, y la segunda es descruzando todos los pares de aristas de la lista de acuerdo a cualquiera de los tres criterios elegido previamente y el algoritmo para cuando la lista L ya no contiene aristas que se están cruzando, lo que equivale a decir que se ha obtenido un circuito hamiltoniano libre de cruces.

Fase II, paso 3: Se obtiene circuito hamiltoniano mejorado (libre de cruces)

Cabe señalar que también se calculó la mejora del circuito hamiltoniano con la técnica 2-opt clásica para llevar a cabo la comparación en tiempo y calidad del tour final, con respecto la variante propuesta. La parte más importante de esta metodología es la identificación de los cruces ya que es la estrategia que ayuda a reducir el espacio de soluciones de la técnica 2-opt clásica.

La metodología propuesta en el peor de los casos tiene un tiempo de ejecución de $O(n^3)$, ya que el paso que lleva el mayor tiempo es el de descruzar a las aristas para obtener un tour libre de cruces , demostrado por Leeuwen y Schoone [164].

A continuación se detalla el proceso para la identificación de las aristas que se intersecan.

5.3 Proceso para la identificación de las aristas que se intersecan

5.3.1 Tipos de cruces en el espacio R2

En el trabajo propuesto por Leeuwen y Schoone [16], se detallan tres tipos de cruces, en los cuales se incluyen todos los posibles casos de intersecciones entre las aristas en un circuito en el plano euclideano:

- Los cruces tipo I o cruces simples consideran los pares de aristas que se intersecan en un solo punto.
- Los cruces tipo II son aquellos en donde una de las aristas se interseca en un vértice de la otra arista.
- Los cruces del tipo III incluyen todas aquellas en donde se traslapa una arista con parte de la otra.

Una sola arista puede tener varias cruces, que además pueden ser de diferentes tipos. Por ejemplo: tres aristas que se cruzan en un solo punto, una arista que se cruza con otras dos aristas traslapadas en un solo punto, etc. Es importante recalcar que al quitar las cruces, se debe volver a reconectar un solo circuito Hamiltoniano: no es válido generar sub-circuitos. El tipo de cruce que más se presenta es el de tipo I.

En la figura 5.2 se pueden observar los tipos de cruce mencionados y su forma de removerlos.

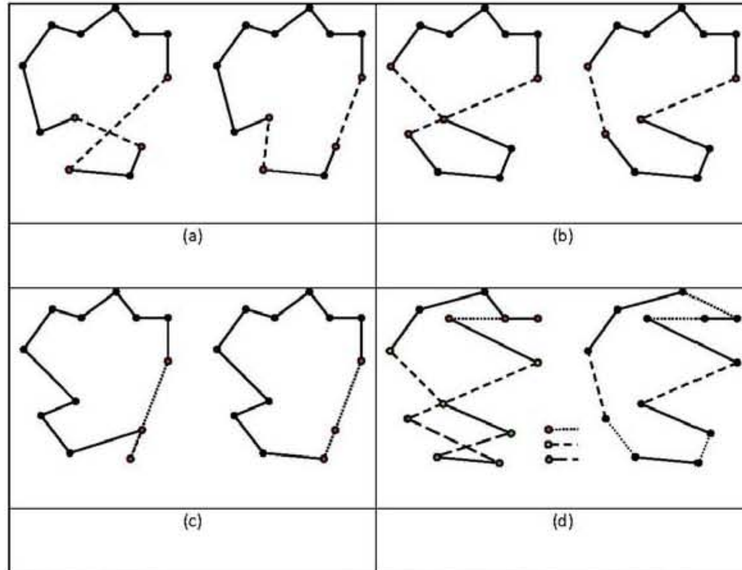


Figura 5.2 Diferentes tipos de cruces y la forma de descruzar. (a) cruce sencillo, (b) intersección de una arista y un vértice, (c) traslape de aristas, (d) combinación de diferentes tipos de cruce en un circuito Hamiltoniano.

En la Figura 5.3 se puede observar las formas correcta e incorrecta de llevar a cabo un descruce para los cruces tipo I.

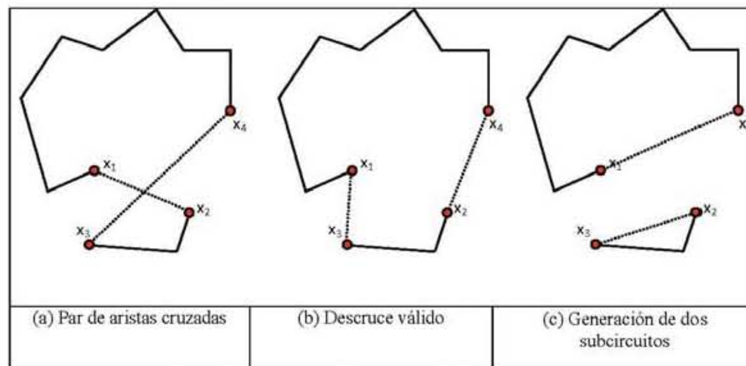


Figura 5.3 Representación de un movimiento de descruce.

Para identificar los pares de aristas que se están intersecando, se realiza lo siguiente:

1. Se consideran ambas aristas (tomadas de la lista L) como rectas, descritas por su ecuación vectorial: $\mathbf{p_i} + \lambda_i * \mathbf{v_i} = 0$ (con i = 1, 2), en donde $\mathbf{p_1}[x_0, y_0]$ y $\mathbf{p_2}[u_0, v_0]$ son los vectores de posición, $\mathbf{v_1}[x, y]$ y $\mathbf{v_2}[u, v]$ los vectores dirección y los parámetros λ_1 y λ_2 son escalares $\in [0 \leq \lambda_1, \lambda_2 \leq 1]$
2. El punto de intersección (λ_1, λ_2) es aquel que pertenece a ambas rectas y en donde:
$$\mathbf{p_1} + \lambda_1 * \mathbf{v_1} = \mathbf{p_2} + \lambda_2 * \mathbf{v_2}$$

   Lo anterior genera un sistema de dos ecuaciones paramétricas:
$$\begin{cases} x_0 + \lambda_1 x = u_0 + \lambda_2 u \\ y_0 + \lambda_1 y = v_0 + \lambda_2 v \end{cases} \quad \text{ó} \quad \begin{cases} \lambda_1 x - \lambda_2 u = u_0 - x_0 \\ \lambda_1 y - \lambda_2 v = v_0 - y_0 \end{cases}$$

3. Se calcula el determinante $\begin{vmatrix} x & u \\ y & v \end{vmatrix}$ para determinar si el sistema es compatible indeterminado o incompatible. Si el determinante es diferente de cero, el sistema es compatible determinado, es decir, existe una solución única. Se continúa al paso 4. En el caso contrario, se para el algoritmo. Regresa al paso 1 para considerar otro par de aristas. Ya que la metodología propuesta considera únicamente los casos compatibles determinados.

4. Se utiliza la regla de Cramer para resolver el sistema lineal anterior de 2x2 para encontrar el conjunto de parámetros (λ_1, λ_2).

5. Los valores de λ_1 y λ_2 indican la posición relativa del cruce con respecto a los puntos extremos. Si los parámetros caen en el rango [0,1], el punto de intersección está contenido en ambas aristas; y se guardan ambas aristas en la lista L, y se regresa al paso 1, si no, las rectas se intersecan en un tramo que no contiene la arista. Valores negativos de λ_1 y λ_2 indican que una arista se encuentra antes del inicio de la otra, valores mayores de 1 indican que la arista se encuentra después del término de la primera. Cabe mencionar que el sentido en el que se toma el vector dirección influye en los valores de los parámetros λ_1 y λ_2; sin embargo, seguirán dentro del rango [0,1] cuando hay una intersección en las aristas.
6. Se obtiene la lista L

5.3.2 Identificación del tipo de cruce

El valor de los determinantes $\begin{vmatrix} x & u \\ y & v \end{vmatrix}, \begin{vmatrix} u_0 & x_0 & u \\ v_0 & y_0 & v \end{vmatrix}$ y $\begin{vmatrix} x & u_0 & x_0 \\ y & v_0 & y_0 \end{vmatrix}$ determina el tipo de cruce que se presenta. Se pueden dar los siguientes casos:

- Si el determinante $\begin{vmatrix} x & u \\ y & v \end{vmatrix}$ es diferente de cero existe un único punto de intersección. En el caso especial en donde λ_1 o λ_2 son exactamente 0 o 1, se trata de un cruce de tipo II; en el caso contrario de un cruce tipo I. Cuando dos aristas son adyacentes λ_1 y/o λ_2 tendrán valores de 0 o 1, sin embargo este caso no se presenta ya que el algoritmo no permite tomar pares de aristas adyacentes.
- Si los tres determinantes $\begin{vmatrix} x & u \\ y & v \end{vmatrix}$, $\begin{vmatrix} u_0 \quad x_0 & u \\ v_0 \quad y_0 & v \end{vmatrix}$ y $\begin{vmatrix} x & u_0 \quad x_0 \\ y & v_0 \quad y_0 \end{vmatrix}$ son iguales a cero, el sistema tiene múltiples soluciones, ya que ambas aristas están ubicados sobre la misma recta. No es posible distinguir con este algoritmo si ambas aristas comparten un tramo (cruce de tipo III) o si pertenecen a dos diferentes tramos de la misma recta. Sin embargo, no influye en la metodología ya que el cruce tipo III establece la posibilidad de que dos ciudades sean visitadas en más de dos ocasiones, lo cual viola una de las restricciones del PAVE.
- Si $\begin{vmatrix} x & u \\ y & v \end{vmatrix}$ es cero, pero $\begin{vmatrix} u_0 \quad x_0 & u \\ v_0 \quad y_0 & v \end{vmatrix}$ y $\begin{vmatrix} x & u_0 \quad x_0 \\ y & v_0 \quad y_0 \end{vmatrix}$ son diferentes de cero, no existe solución, ambas rectas son paralelas y no existe cruce de ningún tipo.

El algoritmo anterior se puede hacer en forma exhaustiva llamado Fuerza Bruta (FB) que implica analizar todas las pares de aristas, o se puede utilizar un algoritmo de geometría computacional como por ejemplo el de Bentley-Ottman [178] que lleva a cabo un barrido de plano para detectar cambios en las posiciones relativas de los vértices para mandar llamar el algoritmo de análisis de cruce solamente cuando existe la posibilidad de detectar uno.

Se comentaba en párrafos anteriores que hay dos formas de proceder al procesar la lista L de cruces detectados. Se puede, en una sola iteración, remover todas los cruces de la lista sin importar los nuevos cruces que se pueden generar, o bien después de cada descruce de un solo par de aristas, volver a generar la lista y elegir de nuevo el mejor candidato a ser descruzado de acuerdo con la regla de selección determinado anteriormente. Dependiendo de la instancia que se analiza, recalcular la lista en cada paso puede doblar o triplicar el tiempo requerido para una iteración completa, obteniendo el mismo circuito final.

5.4 Desarrollo de la metodología propuesta

Una vez identificado el algoritmo de cálculo de pares de aristas que se cruzan, se desarrollaron los cálculos de la metodología propuesta. Esta metodología se resume en los siguientes pasos:

Fase I, Paso 1: Se construye una instancia. Esta puede generarse de manera aleatoria o bien se manda llamar cualquiera de las 74 instancias de la TSPLIB menores a 10,000 ciudades.

Fase I, Paso 2: Se construye un circuito hamiltoniano con cualquiera de las 6 técnicas de construcción, vecino más cercano, mejor vecino más cercano, inserción más barata, más lejana, más cercana e inserción aleatoria, cabe señalar que a las técnicas de inserción se le incorporó la estrategia de casco convexo previo a la actuación del algoritmo.

Fase II, Paso 1: Una vez construido el circuito se procede a identificar las aristas que se cruzan con el algoritmo de Bentley-Ottman y se guardan en una lista L. Este paso es el medular, y propuesta de este proyecto, ya que se reduce el espacio de búsqueda del algoritmo 2-Opt, permitiendo hacer más eficiente en tiempo la solución.

Fase II, Paso 2: Se toman las aristas de la lista L bajo tres criterios de llamado: FIFO, ALEATORIO y PRIORIZADO estas serán las únicas en intercambiarse.

Fase II, Paso 3: Se obtiene circuito hamiltoniano mejorado (libre de cruces)

Figura 5.3 Pseudocódigo del algoritmo propuesto

Se pretende mostrar a través de la experimentación que la inclusión de los elementos de la geometría computacional vuelve más eficientes a las técnicas de construcción como de búsqueda local. Para lograrlo se realizaron las siguientes consideraciones.

5.4.1 Detalle de programación

Los algoritmos se corrieron en un clúster del Instituto de Ciencias Nucleares de la UNAM. El programa en sí no es paralelo, son algoritmos seriales, y se dividieron manualmente el espacio de parámetros que deseamos evaluar, sobre las 6 computadoras del clúster básicamente, cada máquina tiene fijo un algoritmo de construcción y varía el resto de los parámetros.

Se cuenta con cuatro parámetros principales: SOLVERS, FINDERS, RECALCS y CROSSES_ITERS, donde cada parámetro cuenta a su vez con otros parámetros secundarios que pueden variar de acuerdo al cálculo que se requiera. El parámetro SOLVER constituye la solución inicial dada por cualquiera de los 6 algoritmos de construcción, nearest neighbour (nn), best nearest neighbour (bnn), insertion near (in) insertion cheap (ich) ,insertion rand (ir) e insertion far (if). El parámetro FINDER indica el algoritmo mediante el cual se identificaron los cruces, brute force (brute) que significa comparar todos los pares de aristas y el algoritmo Bentley-Otmann (benott) que utiliza la estrategia de geometría computacional del barrido de plano. El parámetro RECALC es una bandera que indica si hay recalculo o no en la identificación de los cruces. El parámetro false indica que no recalcula la lista de pares de aristas por cada iteración y true recalcula los pares de aristas tan pronto se realice un movimiento 2-opt. El parámetro

CROSSES_ITER es el criterio en el que se toman los pares de aristas para descruzarlas. Así, el parámetro FIFO indica que se toma el primer par de aristas de la lista para descruzarlas, ALEATORIO indica que los pares de aristas se toman de la lista de manera aleatoria y PRIORIZADO toma primero el par de aristas que contiene la arista de mayor longitud. Finalmente se obtiene un tour libre de cruces.

Cabe señalar que también se calculó la técnica 2-Opt clásica, una vez construido un circuito hamiltoniano con alguna de las seis técnicas de construcción, para comparar los tiempos de ejecución y calidad del tour.

Se destaca también que la literatura, Fisher [184], Bentley [185], [157] y Jhonson [14] señala que la representación del tour como un árbol simplifica los cálculos requeridos para la actuación de los algoritmos propuestos, en este estudio se emplea un árbol para la representación de un tour o circuito hamltoniano.

Se puede observar que tenemos un espacio de parámetros no trivial, en la figura 5.4, se puede observar cómo juegan los parámetros y la variación en cada caso.

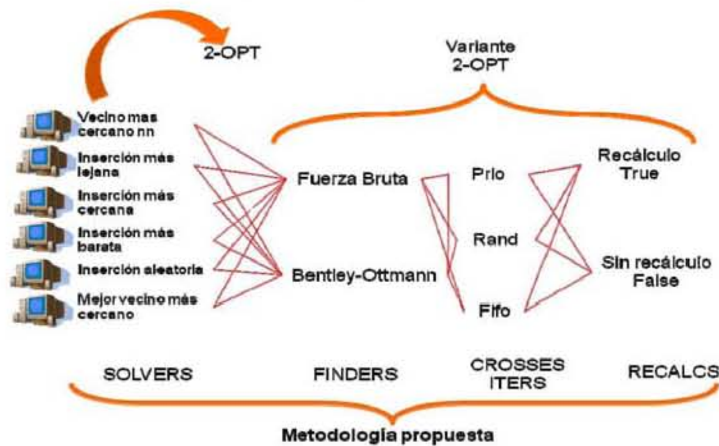
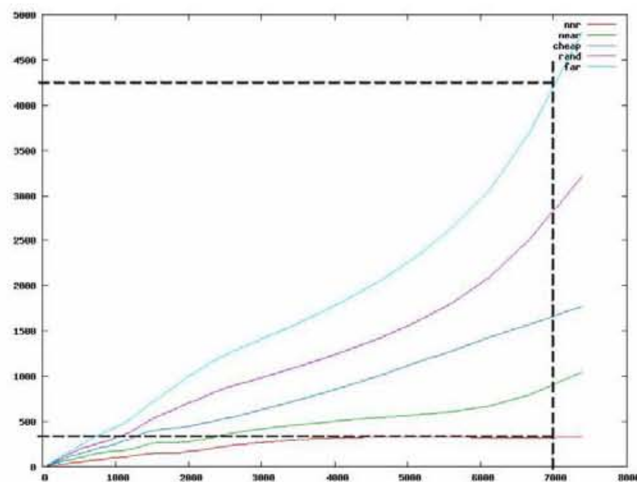


Figura 5.5 Conjunto de parámetros que se calcularon con la metodología propuesta.

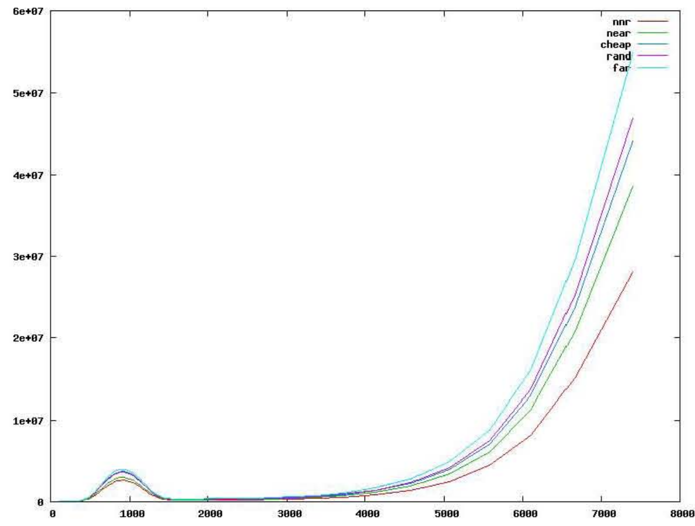
En cada iteración se guarda la siguiente información: tiempo total en segundos, la permutación inicial producida por el algoritmo de construcción, la permutación final producida por el algoritmo de mejora, y la salida del programa que contiene información adicional, como el número de iteraciones, los cruces encontrados, costos de los circuitos iniciales y finales.

5.5 Resultados obtenidos

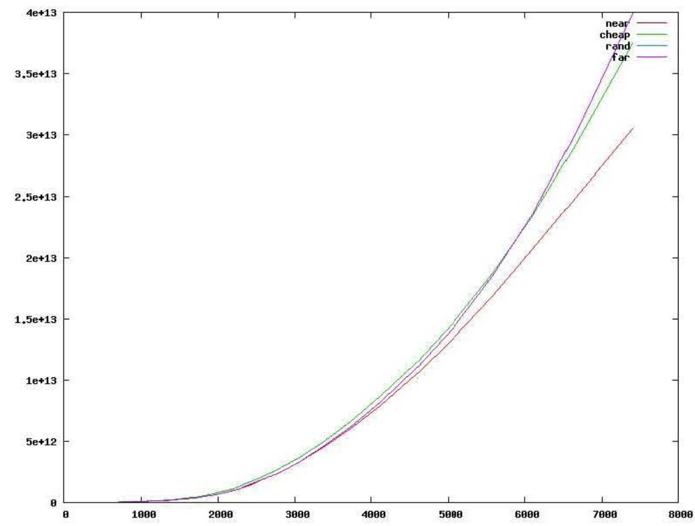
En las gráficas que se presentan a continuación, cabe aclarar que en todas ellas, el eje de las abscisas representa el tamaño de la instancia empleada (número de ciudades), sólo varía el eje de las ordenadas. En la gráfica 5.1 donde el eje de las ordenadas representa el numero de cruces que contienen los circuitos hamiltonianos, se observa que el algoritmo de vecino más cercano (nnr) nos arroja tours con menor cantidad de cruces, por ejemplo en las instancias de 7000 puntos, esta técnica muestra aproximadamente 300 cruces, mientras que con la técnica de inserción con el criterio del más lejano muestra aproximadamente 4700 cruces. Lo cual marca una gran diferencia entre ambas técnicas. En la gráfica 5.2 el eje de las ordenadas representa el costo del tour, y se muestra la calidad del tour arrojada por cada una de las técnicas de construcción, y demuestra la relación directa que hay entre la calidad del tour y el numero de cruces asociados a este. Por ejemplo en las instancias de 7000 puntos se muestra un mejor costo del tour arrojado por la técnica del vecino más cercano mientras que la peor es arrojado por la técnica de inserción más lejana. Lo cual quiere decir que hay una relación directa con la cantidad de los cruces y la calidad del tour, así a menor cantidad de cruces mejor será la calidad del tour.



Gráfica 5.1 Cantidad de cruces generados en cada una de las técnicas de construcción.



Gráfica 5.2 Calidad del tour final arrojado por las técnicas de construcción.

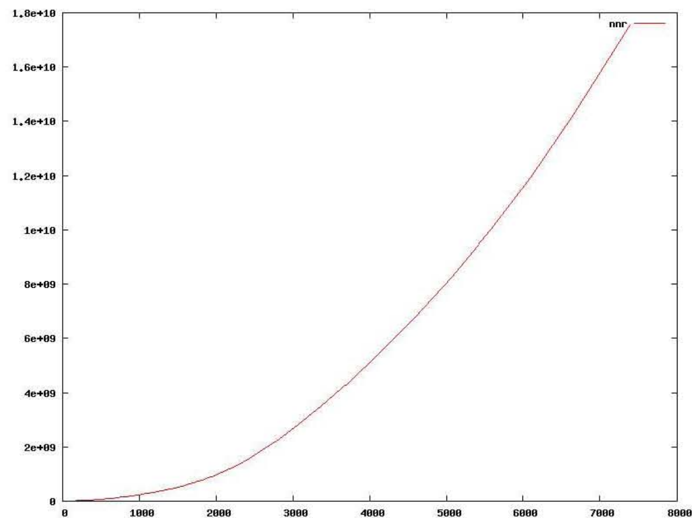


Gráfica 5.3 Tiempo de procesamiento promedio de todas las instancias de la TSPLIB menores a 10,000 puntos con las técnicas de inserción.

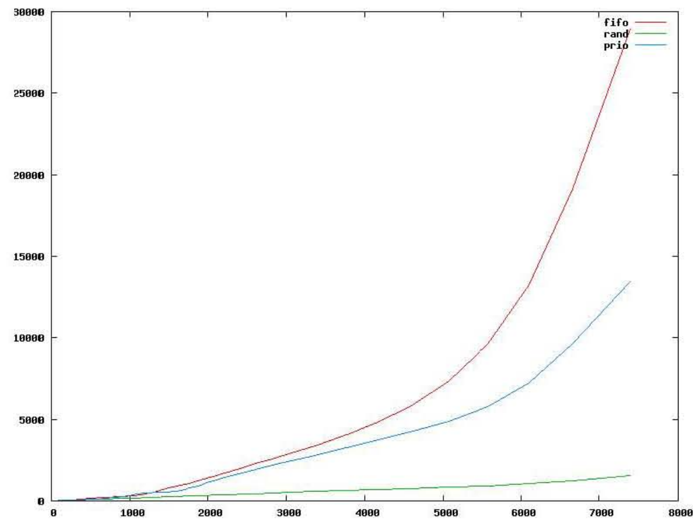
En la gráfica 5.3 se muestra que dentro del conjunto de las técnicas de inserción el que incluye tomar el más cercano (near) es la que nos ofrece el menor tiempo de procesamiento promedio. La gráfica incluye el rango de tiempo desde 0 hasta 40000 nanosegundos (representado por el eje de las ordenadas) que equivale al tiempo total de procesamiento de todas las instancias menores a 10,000 puntos. Por ejemplo, el tiempo de procesamiento para las 49 instancias menores a 1000 puntos el tiempo de procesamiento será menor a 4 minutos.

Mientras que la gráfica 5.4 muestra que el tiempo promedio de procesamiento de la técnica del vecino más cercano en el conjunto de todas las instancias de la lista TSPLIB menores a 10,000 nodos es de 18 segundos.

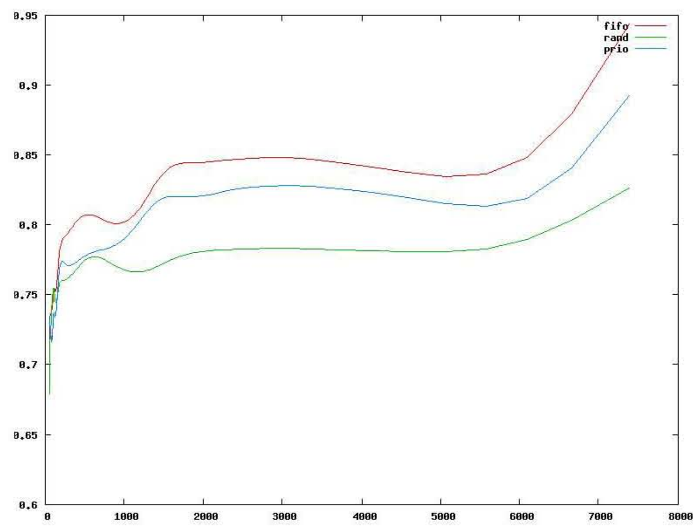
Por otro lado en la gráfica 5.5 en donde el eje de las ordenadas representa el número de iteraciones, muestra que la estrategia de selección al par de aristas de la lista L, ALEATORIO (Rand) es de una diferencia significativa con respecto a las demás, ya que en las instancias superiores a 7000 puntos se requieren 29,000 iteraciones para obtener un circuito libre de cruces, mientras que para la estrategia FIFO, se requieren 1600.



Gráfica 5.4 Tiempo de procesamiento promedio de todas las instancias de la TSPLIB menores a 10,000 puntos con la técnica de vecino más cercano.



Gráfica 5.5 Iteraciones requeridas por los tres criterios de descruce.



Gráfica 5.6 Porcentaje de descruces en cada iteración

En la gráfica 5.6 en donde el eje de las ordenadas representa porcentajes, se muestra el porcentaje de descruces en cada iteración efectuada por los tres criterios de descruce de la lista L. Se observa que el criterio ALEATORIO (rand) es quien en promedio efectúa un 22.5% de descruces en cada iteración, PRIO contará con 17% y FIFO será de un 15%.

La gráfica 5.7 donde el eje de las ordenadas representa porcentajes, muestra el radio promedio de mejora del costo del tour en cada iteración. Y coincidiendo con la gráfica anterior, a mayor porcentaje de descruces logrados por cada iteración se obtiene un costo menor del tour en cada iteración. Se muestra por ejemplo que la estrategia ALEATRORIO (rand) es la de mayor radio de mejora del costo del tour en cada iteración. Sin embargo la diferencia entre los tres criterios hay un .05%.

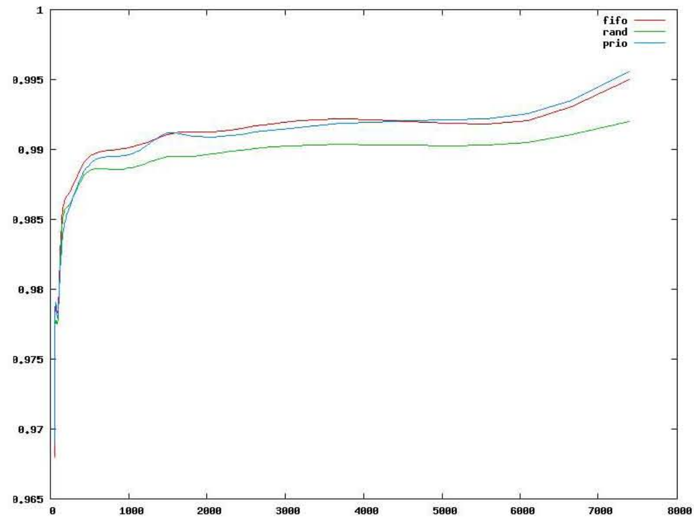
La gráfica 5.8 donde el eje de las ordenadas representa porcentajes , representa la diferencia de mejora del costo del tour expresado en porcentaje, bajo los tres criterios de descruce. En general la estrategia ALEATORIO (rand) es la que presenta el mayor porcentaje de ganancia, un 22%, en el costo del tour final, seguido por la estrategia FIFO con un 21% y PRIORIZADO con un 20%. Se puede observar que las estrategias con la que se descruza el tour no influyen significativamente en la calidad del tour final.

En la gráfica 5.9 donde el eje de las ordenadas representa el tiempo de ejecución promedio, se observa que la estrategia PRIORIZADO tiene el menor tiempo promedio de procesamiento. Seguido por la estrategia FIFO y la de mayor resulta ser ALEATORIZADO. Sin embargo para instancias menores a 1000 esta diferencia es poco significativa.

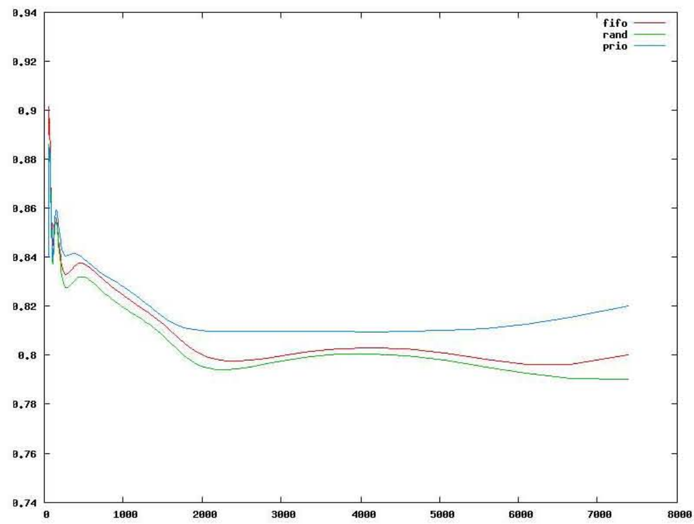
En la gráfica 5.10 donde el eje de las ordenadas representa el número de iteraciones, se observa que cuando se tiene la opción de recalculo en promedio las tres estrategias PRIORIZADO (prio), ALEATORIO (rand) y FIFO, se tiene un promedio de 12 hasta 30,000 iteraciones, dependiendo del número de nodos que tenga la instancia. Mientras que sin recálculo las iteraciones oscilan entre 3 y 10 iteraciones promedio.

En la grafica 5.11 donde el eje de las ordenadas representa porcentajes, se muestra que cuando no se tiene recalculo se descruzan las aristas de la lista L en un mayor porcentaje que cuando se tiene recalculo. Por ejemplo cuando no se recalcula la lista L, para instancias de 1000 puntos se descruzan alrededor del 60% de aristas, mientras que con recalculo es imperceptible el descruce. Y en general se observa que cuando se recalcula la lista L no hay gran porcentaje de descruce en cada iteración. Por lo que se concluye que sin recalculo tengo menos probabilidad de generar cruces.

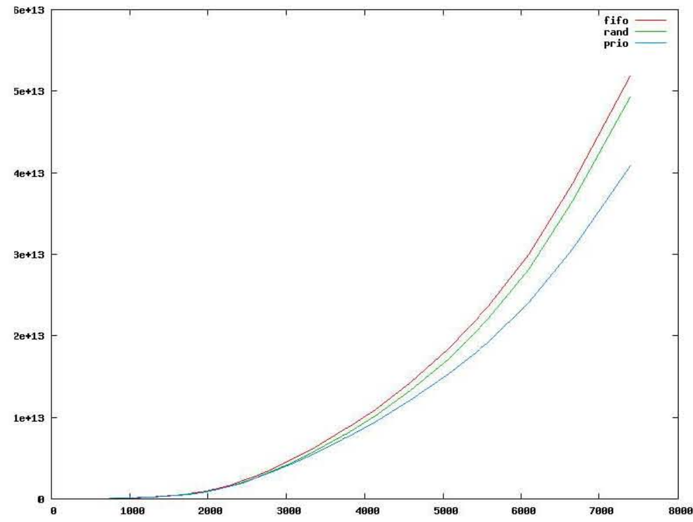
La gráfica 5.12 donde el eje de las ordenadas representa porcentajes, muestra que cuando no se lleva a cabo recalculo se mejora el costo del tour desde un 0.005 en las instancias menores a 1000 puntos hasta un 2% en cada iteración en las instancias entre 7000 y 8000 puntos. Mientras que si se lleva a cabo el recalculo no se nota la mejora en costo del tour.



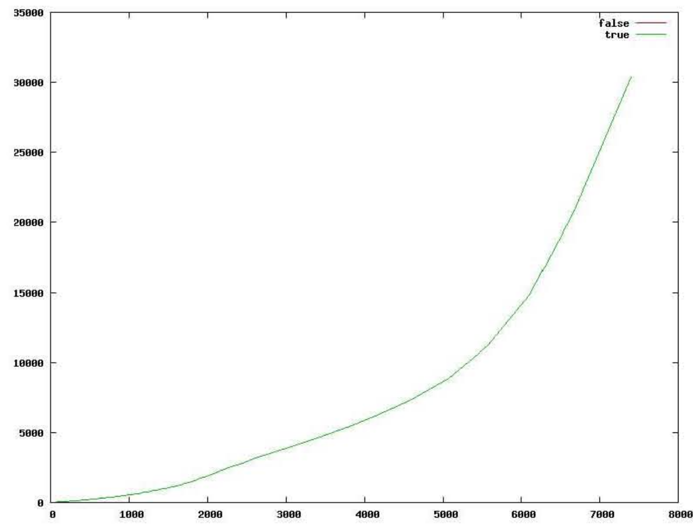
Gráfica 5.7 Radio de mejora en cada iteración en los tres criterios de descruce.



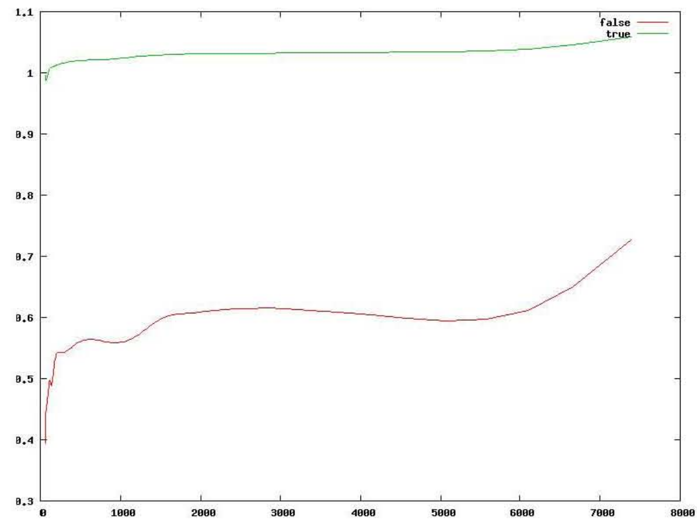
Gráfica 5.8 Radio de mejora total en los costos del tour.



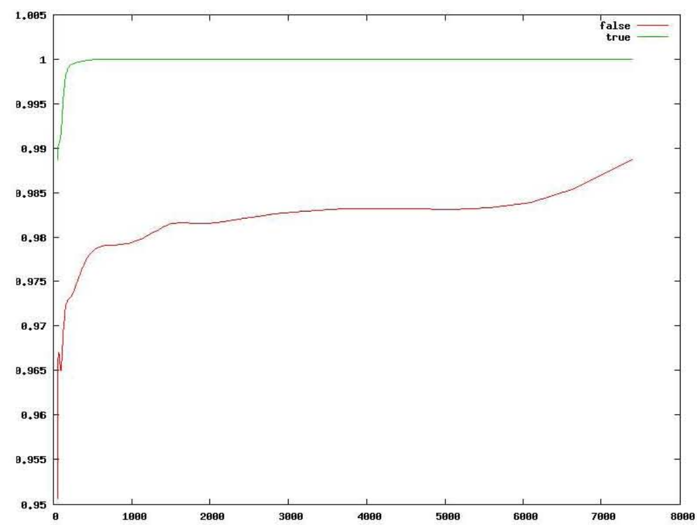
Gráfica 5.9 Tiempo de procesamiento de las estrategias de descruce.



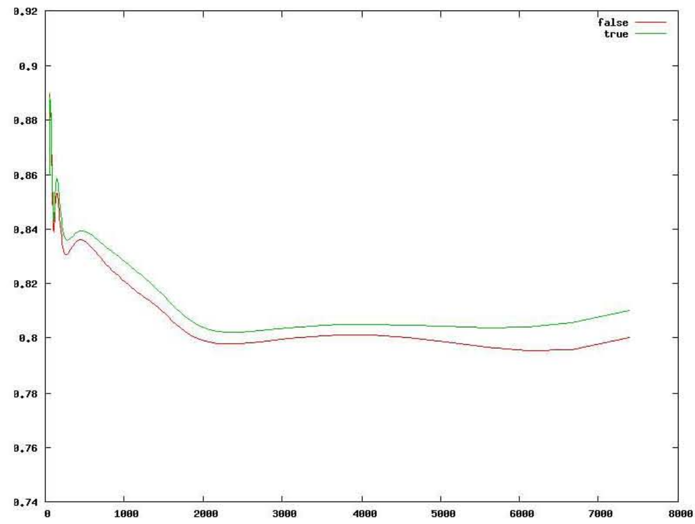
Gráfica 5.10 Tiempo de procesamiento con y sin recalculo de la lista L.



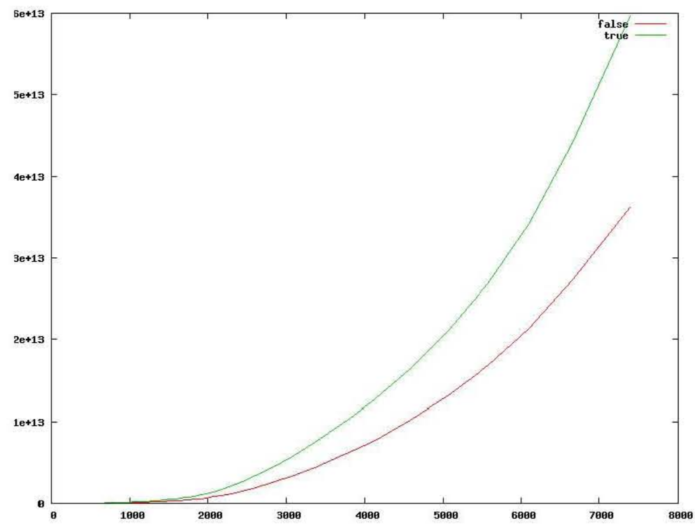
Gráfica 5.11 Porcentaje de descruce por iteración con y sin recalculo.



Gráfica 5.12 Porcentaje de mejora con y sin recalculo.



Gráfica 5.13 Porcentaje de mejora en la calidad del tour final con y sin recalculo.



Gráfica 5.14 Tiempo de procesamiento promedio.

La gráfica 5.13 muestra de manera general que cuando se efectúa el recalculo la mejora en el costo del tour final es mayor cuando no se efectúo el recalculo. Por ejemplo para las instancias de 7000 puntos tengo el 20% de mejora sin recalculo mientras que con recalculo mejoro en un 18% aproximadamente.

En la gráfica 5.14 se observa que el tiempo de procesamiento sin recalculo es aproximadamente 50% menor que si se lleva a cabo con recalculo.

Enseguida se muestran los resultados de la eficiencia de los algoritmos de descruce Bentley-Ottman y Fuerza Bruta.

5.5.1 Medición y resultados de la eficiencia en tiempo de los algoritmos de identificación de pares de aristas que se cruzan.

Una vez ejecutados todos los parámetros así como sus variaciones, se obtuvieron los siguientes resultados de la metodología propuesta. Utilizando la técnica de En la tabla 5.1 se observa que el algoritmo Bentley-Ottman toma ventaja en tiempo sobre el algoritmo de fuerza bruta . La identificación de las aristas que se cruzan con los algoritmos mencionados, son del tipo I.

Instancia	Tiempo Bentley Ottman	Time Fuerza Bruta	Instancia	Tiempo Bentley Ottman	Time Fuerza Bruta	Instancia	Tiempo Bentley Ottman	Time Fuerza Bruta
a280	0.019	0.015	kroB100	0.000	0.012	pr299	0.000	0.018
berlin52	0.000	0.012	kroB150	0.000	0.015	pr439	0.000	0.021
bier127	0.000	0.012	kroB200	0.012	0.012	pr76	0.000	0.012
ch130	0.000	0.020	kroC100	0.000	0.012	rat195	0.012	0.018
ch150	0.000	0.020	kroD100	0.000	0.012	rat575	0.022	0.026
d198	0.002	0.015	kroE100	0.000	0.012	rat783	0.027	0.027
d493	0.003	0.023	lin105	0.000	0.012	rat99	0.012	0.012
d657	0.004	0.025	lin318	0.012	0.017	rd100	0.012	0.015
dsj1000	0.004	0.026	p654	0.019	0.026	rd400	0.018	0.024
eil101	0.000	0.015	pcb442	0.020	0.021	st70	0.012	0.000
eil51	0.000	0.012	pr107	0.000	0.012	ts225	0.012	0.015
eil76	0.001	0.012	pr124	0.012	0.012	tsp225	0.020	0.019

fl417	0.003	0.022	pr136	0.000	0.015	u159	0.012	0.015
gil262	0.002	0.018	pr144	0.000	0.012	u574	0.022	0.024
kroA100	0.001	0.000	pr152	0.012	0.015	u724	0.024	0.025
kroA150	0.001	0.012	pr226	0.015	0.015			
kroA200	0.001	0.012	pr264	0.012	0.020			

Tabla 5.1 Tiempos de ejecución de los algoritmos de identificación de cruces Fuerza Bruta y Bentley-Ottman en el conjunto de instancias menores a 1,000 nodos.

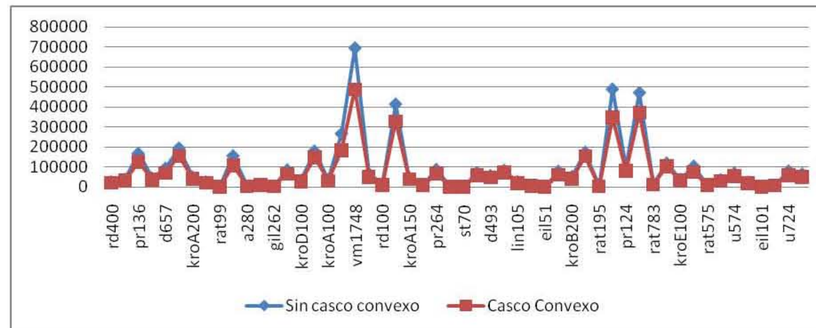
Después de más de 30 años el algoritmo Bentley-Ottman sigue siendo el más popular y el más utilizado en la práctica, ya que es relativamente fácil de entender e implementar. Sin embargo es necesario notar que cuando k (número de intersecciones) es muy grande de orden $O(n^2)$, el algoritmo Bentley-Ottman toma un tiempo de $O(n^2 \log n)$ el cual es peor que el algoritmo de fuerza bruta $O(n^2)$, [178].

Uno puede pensar en implementar el algoritmo de fuerza bruta cuando k se espera que sea más grande que $O(n)$, pero si k se espera ser menor o igual que $O(n)$ se puede utilizar el algoritmo de Bentley-Ottman que se espera tenga en el peor de los casos un tiempo $O(n \log n)$ y en un espacio $O(n)$, [178].

5.5.2 Mejora de las técnicas de inserción con la estructura de casco convexo

La estructura geométrica de casco convexo permite dar una excelente solución inicial a las técnicas de inserción. Wiorkowski y McElvain en 1975[180], Or en 1976[181], Stewart en 1977[182], Norback y Love en 1977[183] afirman que otorgando un sobtour inicial alas técnicas de inserción se tiene como resultado un tour final de mejor calidad.

En la gráfica 5.15, se muestra el comportamiento del costo del tour con y sin la estructura de casco convexo como solución inicial a las técnicas de inserción con el cirterio de inserción aleatoria, mostrando una mejora en el tour en un 6% . Los resultados para los criterios restantes muestran que la inclusión de la estructura de casco convexo mejora en un 6% la calidad del tour con los criterios del más cercano y el más lejano, y 5% con el criterio de más barato.



Gráfica 5.15 Calidad del tour de la técnica de inserción con el criterio de aleatorio, con y sin casco convexo como solución inicial.

5.6 Resumen de resultados de calidad y tiempo de solución con respecto a la solución óptima

En las siguientes dos tablas 5. 3 y 5.4 se muestran los porcentajes de desviación de la longitud obtenida con respecto al valor óptimo, de todas las instancias de la TSPLIB menores a 1,000 puntos, ya que se concentran en este rubro la mayoria de las instancias, también se se emplearon las tres estrategias distintas para descruzar los pares de aristas y las seis técnicas de construcción, sin el empleo de la estructura de casco convexo, ya que los resultados muestran que el tour final Los resultados se obtuvieron sin recalcular la lista L , ya que como muestra la gráfica 5.10, el tiempo con recálculo implica mayor inversión en tiempo, obteniéndose el mismo resultado que sin recálculo y se ocupó el algoritmo de Bentley-Ottman para la identificación de las aristas que se intersecan.

Heuristica	N.N. (Vecino más cercano)		Inserción más cercana		B.N.N. (Mejor vecino más cercano)	
	% Desviación	Tiempo Ejecución Seg	% Desviación	Tiempo Ejecución Seg	% Desviación	Tiempo Ejecución Seg
FIFO	16.65%	0.63	12.49%	0.24	10.93%	0.38
RAND	16.06%	0.71	12.47%	0.27	11.07%	0.5
PRIO	16.76%	0.57	12.45%	0.25	11.19%	0.56

Tabla 5. 3 Tiempo y calidad de solución de las técnicas que superan el 10% de desviación con respecto al valor óptimo.

En la tabla 5.3 se tienen los tres algoritmos que superan un porcentaje de desviación con respecto al valor óptimo en un 10%, asi la técnica del vecino más cercano, incersión más cercana y el mejor vecino más cercano, son algoritmos que arrojan tours que no son un buena combinación con la variante propuesta.

Heuristica	Inserción Aleatoria		Inserción más barata		Inserción más lejana	
	% Desviación	Tiempo Ejecución Seg	% Desviación	Tiempo Ejecución Seg	% Desviación	Tiempo Ejecución Seg
FIFO	7.27%	0.89	8.85%	0.16	8.05%	0.79
RAND	9.00%	0.1	8.85%	0.13	8.05%	0.72
PRIO	9.00%	0.11	8.86%	0.14	8.05%	0.74

Tabla 5.4 Tiempo y calidad de solución de las técnicas que se encuentran por debajo del 9% de desviación con respecto al valor óptimo.

En la tabla 5.4 se observan los algoritmos que arrojan tours cuyo porcentaje de desviación con respecto al valor óptimo son inferiores a un 9%. La mejor combinación es construir un circuito con el algoritmo de inserción aleatoria y el descruce con la estrategia FIFO (primero que entra primero que sale) con un porcentaje de desviación con respecto al valor óptimo del 7.27%.

2-Opt con	N.N. (Vecino más cercano)	Inserción más cercana	Inserción más lejana	Inserción más barata	Inserción aleatoria	B.N.N. (Mejor vecino más cercano)
% Desviación	7.92%	7.61%	6.78%	6.15%	6.15%	6.19%
Tiempo Ejecución Seg	3.69	1.72	3.81	1.2	1.2	2.8

Tabla 5.5. Tiempo y calidad de solución de la técnica 2-Opt clásica con seis distintos técnicas de construcción.

En la tabla5. 5 se muestran los porcentajes de desviación con respecto al valor óptimo y los tiempos de ejecución de la técnica 2-Opt clásica (propuesta por Croes) en combinación con las seis técnicas de construcción distintas. Se observa que las técnicas de inserción aleatoria (RAND) y las más barata (Cheap) son las técnicas de construcción que mejor comportamiento tienen en combinación con la técnica 2-Opt clásica, arrojando porcentajes de desviación con respecto al valor óptimo de 6.15% en ambas técnicas.

A continuación, en la tabla 5.6, se muestra un comparativo de el valor de la función objetivo y el tiempo de ejecución de la metodología propuesta con respecto a la técnica 2-opt clásica, es decir, la propuesta por Croes. La metodología propuesta utiliza la mejor combinación que se obtuvo en el análisis del conjunto de instancias de la TSPLIB menores a 1000 ciudades y se aplicó esta metodología en el conjunto de instancias menores a 10,000. Esta metodología construye el circuito hamiltoniano con la técnica de inserción aleatoria y se disminuye la vecindad identificando los pares de aristas que se intersecan con el algoritmo de Bentley-Ottman y descruzando el primer par de aristas que se intersecan es decir, la estrategia FIFO. En resumen se obtiene que utilizando la metodología propuesta se logra un 8.42% de desviación con respecto al valor óptimo en un tiempo de ejecución de 12.68 segundos, mientras que la técnica 2-Opt propuesta por Croes presenta 6.15% de desviación con respecto al valor óptimo en un tiempo de 30.82 segundos. Estos tiempos no son normalizados y se utilizaron todas las instancias de la TSPLIB menores a 10,000 ciudades.

Nombre Instancia TSPLIB	Número de ciudades	Función objetivo óptimo	Función objetivo esther	Función objetivo 2-Opt	Desviación óptimo esther	Desviación óptimo 2-Opt	Tiempo Esther	Tiempo 2-Opt	Normalizado Esther	Normalizado 2-Opt
a280	280	2579	2796	2745	8.41%	6.44%	0.15	1.01	0.00	0.01
berlin52	52	7542	8160	8002	8.19%	6.10%	0.01	0.05	0.00	0.00
bier127	127	118282	128121	125519	8.32%	6.12%	0.05	0.31	0.00	0.01
ch130	130	6110	6621	6477	8.36%	6.01%	0.06	0.40	0.00	0.01
ch150	150	6528	7059	6933	8.13%	6.20%	0.09	0.57	0.00	0.01
d198	198	15780	17056	16751	8.09%	6.15%	0.27	2.22	0.00	0.03
d1291	1291	50801	55063	53922	8.39%	6.14%	29.28	39.38	0.07	0.09
d493	493	35002	37911	37161	8.31%	6.17%	3.18	21.31	0.02	0.12
d657	657	48912	52926	51979	8.21%	6.27%	6.59	48.96	0.03	0.21
d1655	1655	62128	67290	65959	8.31%	6.17%	6.53	49.46	0.01	0.10
dsj1000	1000	18659688	20153574	19871069	8.01%	6.49%	11.89	77.15	0.04	0.23
d2103	2103	80450	87211	85386	8.40%	6.14%	43.89	91.89	0.07	0.15
eil101	101	629	671	668	6.68%	6.20%	0.03	0.14	0.00	0.00
eil51	51	426	459	452	7.75%	6.10%	0.03	0.14	0.00	0.00
eil76	76	538	582	572	8.18%	6.32%	0.01	0.06	0.00	0.00
fl417	417	11861	12891	12598	8.68%	6.21%	1.66	10.86	0.01	0.07
fl1400	1400	20127	21817	21366	8.40%	6.16%	26.89	67.42	0.06	0.15
fl1577	1577	22249	24124	23627	8.43%	6.19%	53.43	78.97	0.11	0.16
fl3795	3795	28772	31160	30531	8.30%	6.11%	43.34	92.73	0.04	0.09
fnl4461	4461	182566	197969	193829	8.44%	6.17%	53.89	94.73	0.04	0.08
gil262	262	2378	2579	2529	8.45%	6.35%	0.37	2.46	0.00	0.02
kroA100	100	21282	23079	22600	8.44%	6.19%	0.03	0.19	0.00	0.00
kroA150	150	26524	28660	28156	8.05%	6.15%	0.13	0.94	0.00	0.01

kroA200	200	29368	31980	31169	8.89%	6.13%	0.19	1.24	0.00	0.01
kroB100	100	22141	23912	23492	8.00%	6.10%	0.06	0.44	0.00	0.01
kroB150	150	26130	28321	27729	8.38%	6.12%	0.12	0.80	0.00	0.01
kroB200	200	29437	31912	31241	8.41%	6.13%	0.09	0.74	0.00	0.01
kroC100	100	20749	22599	22070	8.92%	6.37%	0.03	0.18	0.00	0.00
kroD100	100	21294	23198	22583	8.94%	6.05%	0.02	0.19	0.00	0.00
kroE100	100	28068	30589	29758	8.98%	6.02%	0.07	0.46	0.00	0.01
lin105	105	14379	15628	15258	8.69%	6.11%	0.02	0.11	0.00	0.00
lin318	318	42029	45671	44609	8.67%	6.14%	0.25	1.83	0.00	0.01
nrw1379	1379	56638	61456	60221	8.51%	6.33%	16.31	108.76	0.04	0.25
p654	654	34643	37489	36789	8.22%	6.19%	0.74	4.99	0.00	0.02
pcb1173	1173	56892	61673	60370	8.40%	6.11%	12.17	88.05	0.03	0.23
pcb442	442	50778	54890	53890	8.10%	6.13%	1.48	9.39	0.01	0.06
pcb3038	3038	137694	149694	146289	8.71%	6.24%	33.28	91.30	0.04	0.10
pla7397	7397	23260728	25224929	24695962	8.44%	6.17%	79.69	118.87	0.04	0.06
pr1002	1002	259045	280768	274850	8.39%	6.10%	0.99	6.45	0.00	0.02
pr107	107	44303	48013	47001	8.37%	6.09%	0.03	0.19	0.00	0.00
pr124	124	59030	64100	62572	8.59%	6.00%	0.10	0.68	0.00	0.01
pr136	136	96772	104600	102676	8.09%	6.10%	0.09	0.75	0.00	0.01
pr144	144	58537	63300	62119	8.14%	6.12%	0.07	0.49	0.00	0.01
pr152	152	73682	79871	78179	8.40%	6.10%	0.24	1.49	0.00	0.02
pr226	226	80369	86891	85301	8.12%	6.14%	0.13	0.82	0.00	0.01
pr2392	2392	378032	409410	401169	8.30%	6.12%	3.39	21.97	0.00	0.03
pr264	264	49135	53101	52149	8.07%	6.13%	0.08	0.50	0.00	0.00
pr299	299	48191	52071	51146	8.05%	6.13%	0.51	3.29	0.00	0.03
pr439	439	107217	120996	113780	12.85%	6.12%	6.25	40.75	0.04	0.25
pr76	76	108159	117100	114737	8.27%	6.08%	0.04	0.35	0.00	0.01
rat195	195	2323	2521	2460	8.52%	5.90%	0.64	4.18	0.01	0.05
rat575	575	6773	7329	7196	8.21%	6.25%	10.47	68.27	0.05	0.33
rat783	783	8806	9580	9361	8.79%	6.30%	1.06	7.12	0.00	0.03
rat99	99	1211	1311	1285	8.26%	6.11%	0.02	0.18	0.00	0.00
rd100	100	7910	8580	8394	8.47%	6.12%	0.02	0.13	0.00	0.00
rd400	400	15281	16510	16214	8.04%	6.11%	3.85	27.91	0.03	0.18
rl1304	1304	252948	274183	268413	8.40%	6.11%	4.81	35.57	0.01	0.08
rl1323	1323	270199	292995	286839	8.44%	6.16%	53.89	93.40	0.13	0.22
rl1889	1889	316536	343141	335979	8.41%	6.14%	56.65	87.62	0.10	0.15
rl5915	5915	565530	613198	600339	8.43%	6.16%	53.20	97.28	0.03	0.06
rl5934	5934	556045	602792	590118	8.41%	6.13%	63.14	98.27	0.04	0.06
st70	70	675	731	716	8.30%	6.07%	0.01	0.09	0.00	0.00
ts225	225	126643	137774	134375	8.79%	6.11%	0.58	3.86	0.01	0.04
tsp225	225	3916	4260	4157	8.78%	6.15%	0.42	2.82	0.00	0.03

u1060	1060	224094	246021	237821	9.78%	6.13%	16.22	107.44	0.05	0.31
u1432	1432	152970	165299	162418	8.06%	6.18%	6.47	42.06	0.01	0.09
u159	159	42080	45448	44659	8.00%	6.13%	0.03	0.19	0.00	0.00
u574	574	36905	40101	39176	8.66%	6.15%	0.66	4.33	0.00	0.02
u724	724	41910	45289	44509	8.06%	6.20%	6.52	42.39	0.03	0.17
u1817	1817	57201	62015	60702	8.42%	6.12%	51.77	81.88	0.09	0.15
u2152	2152	64253	69649	68206	8.40%	6.15%	43.44	93.40	0.07	0.14
u2319	2319	234256	253933	248650	8.40%	6.14%	43.90	92.05	0.06	0.13
vm1084	1084	239297	258709	253954	8.11%	6.13%	69.92	73.34	0.20	0.21
vm1748	1748	336556	364810	357264	8.40%	6.15%	23.69	79.36	0.04	0.15
Promedio					**0.0842**	**0.0615**	**12.68**	**30.82**	**0.02**	**0.07**

Tabla 5. 6 Comparación del tiempo y calidad de solución del conjunto de instancias menores a 10,000 ciudades del conjunto de instancias de la TSPLIB, con la metodologia propuesta y el algoritmo 2-Opt de Croes.

En general, la metodología propuesta, que incorpora algunos elementos de la geometría computacional como el casco convexo y la intersección de segmentos de recta, con el paradigma de barrido de plano, logra en promedio una disminución varias órdenes de magnitud menores en tiempo de ejecución con respecto a la técnica 2-Opt clásica, aunque el costo del tour es ligeramente mayor. Las mejores técnicas de construcción resultaron ser las técnicas de inserción en especial con el criterio de más lejano e inserción aleatoria, con el criterio FIFO en el caso de la primera y cualquier estrategia para el caso de la segunda, con 7.27% y 8.05% porcentajes de desviación con respecto al valor óptimo respectivamente, para el conjunto de instancias menores a 1000 ciudades. Mientras que para todas las instancias menores a 10,000 ciudades se obtiene un 8.42% de desviación con respecto al valor óptimo, con la estrategia FIFO y utilizando la técnica de construcción inserción aleatoria. De igual manera, se tiene que, las técnicas de vecino cercano y mejor vecino más cercano se logran resultados del 16% y 12% porcentajes de desviación con respecto al valor óptimo respectivamente, con lo cual se puede inferir que la técnica de construcción usada para generar los circuitos iniciales influyen fuertemente en la eficiencia de la metodología propuesta así como en la técnica 2-Opt clásica, esto ocurre en el conjunto de instancias inferiores a 1000 ciudades. Esto se entiende porque ambas estrategias tienen la misma filosofía, que es el intercambio de aristas. Las diferentes formas de priorización no tuvieron influencia significativa en los resultados del tiempo de ejecución, ni de porcentaje de desviación del circuito final.

A continuación se expone un estudio comparativo con nueve algoritmos (incluyendo la variante (Esther) y la técnica 2-Opt clásica (Croes)). Cada uno de estos algoritmos propone un espacio de búsqueda modificando el tamaño de vecindad. Por ejemplo el algoritmo 2-Opt Bentley (columna siete), utiliza la estructura de dato *"Fixed Radius Near Neighbor "* (FRNN) propuesta por Bentley en 1992 , [157] . La cual consiste en intercambiar los pares de aristas que se encuentran más cercanos a una ciudad en especifico, garantizando que el intercambio obtendrá una mejora. Son estrategias que evitan la redundancia en el espacio

de búsqueda, o bien ordenan el conjunto de aristas de acuerdo al peso de estas y se toman las *k* aristas de esta lista, y en representar al tour como un árbol, tal como se llevó a cabo en este proyecto.

El conjunto de instancias que se emplearon para la comparación se describen en la tabla 5.7

Nombre Instancia TSPLIB	Numero de ciudades	Función objetivo óptimo	Función objetivo esther	Función objetivo 2-Opt	Desviación óptimo esther	Desviación óptimo 2-Opt	Tiempo Esther Intel Pentium IV, a 3GHy con 1 Gb en RAM	Tiempo 2-Opt Intel Pentium IV, a 3GHy con 1 Gb en RAM	Normalizado Esther	Normalizado 2-Opt
d1291	1291	50801	55063	53922	8.39%	6.14%	29.28	39.38	0.07	0.09
d2103	2103	80450	87211	85386	8.40%	6.14%	43.89	91.89	0.07	0.15
fl1400	1400	20127	21817	21366	8.40%	6.16%	26.89	67.42	0.06	0.15
fl1577	1577	22249	24124	23627	8.43%	6.19%	53.43	78.97	0.11	0.16
fl3795	3795	28772	31160	30531	8.30%	6.11%	43.34	92.73	0.04	0.09
fnl4461	4461	182566	197969	193829	8.44%	6.17%	53.89	94.73	0.04	0.08
pcb3038	3038	137694	149694	146289	8.71%	6.24%	33.28	91.30	0.04	0.10
pla7397	7397	23260728	25224929	24695962	8.44%	6.17%	79.69	118.87	0.04	0.06
rl1323	1323	270199	292995	286839	8.44%	6.16%	53.89	93.40	0.13	0.22
rl1889	1889	316536	343141	335979	8.41%	6.14%	56.65	87.62	0.10	0.15
rl5915	5915	565530	613198	600339	8.43%	6.16%	53.20	97.28	0.03	0.06
rl5934	5934	556045	602792	590118	8.41%	6.13%	63.14	98.27	0.04	0.06
u1817	1817	57201	62015	60702	8.42%	6.12%	51.77	81.88	0.09	0.15
u2152	2152	64253	69649	68206	8.40%	6.15%	43.44	93.40	0.07	0.14
u2319	2319	234256	253933	248650	8.40%	6.14%	43.90	92.05	0.06	0.13
vm1748	1748	336556	364810	357264	8.40%	6.15%	23.69	79.36	0.04	0.15
Promedio					**8.43%**	**6.15%**	**47.09**	**87.41**	**0.07**	**0.11**

Tabla 5.7 Conjunto de instancias de la TSPLIB con el que se comparó la metodología propuesta respecto a 8 algoritmos 2-Opt modificados.

Los algoritmos con los que se realiza el comparativo se corrieron en una máquina Compaq ES40 Alpha y los experimentos de la investigación se corrieron en una máquina Intel Pentium IV. La tabla 5.8 está compuesta en dos secciones, en la primera se muestra el porcentaje de desviación promedio con respecto al valor óptimo, destacando que el mejor porcentaje es el algoritmo 2opt-JM-40-quadrant-neighbors (quinta columna) con un porcentaje de 5.91%, mientras que la variante propuesta queda en un lugar promedio con un 8.43%. Sin embargo, los tiempos de ejecución normalizados muestran que la variante propuesta (Esther) tiene un tiempo de ejecución de 0.07 segundos mientras que los algoritmos restantes tienen un tiempo de ejecución de 0.1 segundos, lo cual resulta un tiempo inferior a los otros algoritmos, y en especial a la técnica 2-Opt clásica (Croes) (con

un tiempo de .11 segundos), objetivo primordial de este proyecto de investigación. En general, la variante propuesta muestra ser más rápidos en tiempos de ejecución con respecto los algoritmos restantes.

	2opt-JM-10-quadrant-neighbors wiht don't-look bits	2opt-JM-20-quadrant-neighbors[Alpha] with don't-look bits	2opt-JM-20-quadrant-neighbors[MIPS] Algorithm	2opt-JM-40-quadrant-neighbors[Run2] Algorithm	2opt-JM-40-quadrant-neighbors[Run1] Algorithm	2-Opt:Bentley Algorithm	Concorde-2opt Algorithm	2-Opt Esther	2-Opt Croes
Porcentaje de exceso promedio con el tour óptimo									
Compaq ES40 Alpha	9.74	6.62	6	5.91	6.4	6.88	16.66	-	-
Intel Pentium IV	-	-	-	-	-	-	-	8.43	6.15
Tiempo promedio en segundos promedio (Normalizado NlogN)									
Compaq ES40 Alpha	0.1	0.1	0.1	0.2	0.2	0.1	0.1	-	-
Intel Pentium IV	-	-	-	-	-	-	-	0.06	0.11

Tabla 5.8 Comparativo con diversas estrategias que modifican a la técnica 2-Opt

Conclusiones

Síntesis de resultados

Conclusiones finales del desarrollo del proyecto y futura investigación

---

Se puede concluir que se cumplieron todos y cada uno de los objetivos planteados. Se logra proponer una metodología eficiente en tiempo y calidad de solución que resuelve el problema del agente viajero euclideano, que consiste en modificar el espacio de búsqueda de la técnica de búsqueda local *2-Opt*, así como la mejora de las técnicas de construcción por inserción, a través de la inclusión de elementos de Geometría Computacional, en el caso particular se prueba la metodología en el conjunto de instancias de la TSPLIB.

Para alcanzar este objetivo general se analizó la taxonomía del problema del agente viajero general y euclideano para detectar las propiedades *naturales* con las que cuenta, así como el análisis de los algoritmos de construcción y de búsqueda local y facilitar la inclusión de los elementos de la geometría computacional. Se detectó a través de la literatura que el PAVE cumple de manera *natural* con la desigualdad del triángulo lo cual implica que siempre podremos acortar los tours dirigiéndonos de manera directa a una ciudad sin pasar por ciudades intermedias, mejorando siempre la calidad del tour.

Los algoritmos de búsqueda local altamente prometedores tanto en tiempo y calidad de solución son los que pertenecen a la familia Lin-Kernighan, y dentro de este conjunto se encuentra la técnica *k-Opt* y el caso especial *2-Opt*.

La metodología propuesta consiste en construir un circuito hamiltoniano, mejorarlo y entregar como solución final un circuito libre de cruces. Para lograr una metodología que es eficiente en tiempo y calidad de solución se tomó como base el algoritmo de búsqueda local *2-Opt* quien es una de las más poderosas, y de las cuales aún sigue limitado el análisis teórico, empírico y experimental.

Con base en el análisis de la literatura, los expertos proponen varios aspectos que se tienen que considerar para lograr mayor velocidad en el tiempo de corrida del algoritmo *2-Opt*, entre ellos y los de extrema importancia son: la técnica de construcción utilizada, la estrategia de selección de los pares de aristas a intercambiar que modifica el tamaño de vecindad, lo cual implica generar una lista candidata de aristas que ayuden a disminuir la vecindad del espacio de búsqueda, y los detalles de implementación de la heurística.

Estos aspectos que se tienen que tomar en cuenta para hacer más eficiente al algoritmo *2-opt*, y por consiguiente la metodología propuesta, se implementaron a través de un conjunto de parámetros y programados en el lenguaje de programación JAVA 5, esto permite determinar qué combinación de estos devuelve un tour eficiente tanto en tiempo como calidad de solución. Estos parámetros son: seis técnicas de construcción, vecino más cercano, mejor vecino más cercano, y tres técnicas de inserción con tres criterios distintos más cercano, más lejano, y más barato, un algoritmo que utiliza la estructura de Geometría Computacional *intersección* que permite la identificación de pares de aristas que se intersecan y que permite construir la lista candidata, reduciendo el tamaño de vecindad, utilizando dos paradigmas: *barrido de plano* y *fuerza bruta*, se tiene también tres

estrategias distintas de elegir a los elementos de la lista candidata (que contiene únicamente pares de aristas que se intersecan): FIFO (primer par de aristas que se encuentra en la lista), PRIORIZADO (que consiste en tomar el par de aristas que contenga la de mayor longitud), y ALEATORIO (que toma cualquier par de la lista) y además de dos criterios de calcular esta lista, con recalculo o sin recalculo. Lo cual significa que una vez descruzado un par de aristas con cualquier criterio de llamado se puede volver a calcular o no esta lista.

El corazón de esta metodología consiste en hacer más eficiente la lista candidata de aristas que se tomarán para *borrar* e *intercambiar*, que es la filosofía del algoritmo *2-Opt*. En este sentido la literatura muestra que los criterios de construir esta lista ha variado, utilizando estructuras de elementos de la Geometría Computacional principalmente , sin embargo, ninguna de ellas ha implementado la estructura *Intersección*, y se emplea para esta estructura el paradigma de *barrido de plano* para hacer más eficiente la búsqueda de los pares de segmentos que se intersecan. El criterio de paro del algoritmo, es decir, que el algoritmo terminará su ejecución en algún momento, fue demostrado por, Leeuwen y Schoone [164], quienes demuestran que eliminar todas las intersecciones en una gráfica, densa y completa aún cuando sean insertados nuevos cruces en el proceso, se logra obtener en un tiempo $O(n^3)$.

Como resultado del análisis de los parámetros se obtuvo lo siguiente: en cuanto la técnica de construcción utilizada, se observa que la inclusión de la estructura de casco convexo mejora en un 6% la calidad del tour con los criterios del más lejano, aleatoria, cercano y el más lejano, 5% con el criterio de más barato.

Las técnicas de construcción que no resultan deseables en combinación con la metodología propuesta son, las técnicas del vecino más cercano y su variante el mejor vecino más cercano e inserción más cercana, ya que la calidad del tour final supera el estándar propuesto por Johnson del 10% de desviación con respecto el valor óptimo, y las técnicas de inserción con el criterio de aleatorio, más lejana y más barato, arrojan tours finales inferiores a un 9%. Resultando que la mejor técnica de construcción con la metodología propuesta es la **técnica de inserción con el criterio del más lejano**, con una desviación del óptimo del 8.42% en un tiempo de ejecución de 12.68 segundos, utilizando la **estrategia FIFO** (se toma el primer par de aristas como criterio para descruzar) y se disminuye la vecindad identificando los pares de aristas que se intersecan con el algoritmo de **Bentley-Ottman** en el conjunto de todas las instancias menores a 10000 ciudades de la TSPLIB.

Si se toma la misma la técnica de inserción con el criterio del más lejano en el algoritmo 2-Opt propuesto por Croes se tiene un 6.15% de desviación con respecto al valor óptimo en un tiempo de 30.82 segundos.

En cuanto la estrategia de selección de los pares de aristas a intercambiar que modifica el tamaño de vecindad, lo cual implica generar una lista candidata de aristas que ayuden a disminuir la vecindad del espacio de búsqueda, se identifican los pares de aristas que se intersecan resultando que con el paradigma de *barrido de plano* se encuentran con 0.010 segundos en promedio con mayor velocidad que cuando se utiliza *fuerza bruta*, tomando mayor relevancia en las instancias con más de 500 ciudades con 0.015 segundos y 0.020 más veloz en instancias que superan las 2000 ciudades.

La metodología propuesta se compara con ocho variaciones de la técnica *2-Opt* que se han desarrollado a lo largo de estos últimos años y que fueron expuestos en el 8th DIMACS Implementation Challenge organizado por David Johnson, utilizando solo un conjunto de instancias de la TSPLIB. Estas técnicas utilizan diversas estrategias que modifican la búsqueda en el espacio de soluciones para permitir a la técnica *2-Opt* ser más eficiente, obteniéndose que, en cuanto la calidad del tour se logra un porcentaje de desviación con respecto al valor óptimo de 8.43%, contrastando con la propuesta realizada por Jhonson y McGeoch (2opt-JM-40-quadrant-neighbors) con 5.91%, existiendo resultados más pobres como la propuesta por el Concorde con un 16.6%. Sin embargo, los tiempos de ejecución normalizados muestran que la metodología propuesta tiene un tiempo de ejecución de 0.06 segundos mientras que los algoritmos restantes tienen un tiempo de ejecución de 0.1 segundos, lo cual resulta un tiempo inferior a los otros algoritmos, y en especial a la técnica 2-Opt clásica (propuesta por Croes), con un tiempo de .11 segundos, objetivo primordial de este proyecto de investigación. El resultado más relevante de este proyecto es que en general, la variante propuesta muestra ser el doble de rápida en tiempos de ejecución con respecto a todos los algoritmos restantes.

Futura Investigación

La metodología propuesta incluye las técnicas de construcción más frecuentemente utilizadas para resolver el PAVE, sin embargo, se pueden implementar otras técnicas y determinar su comportamiento. También se pueden insertar combinaciones de los elementos de geometría computacional con diversos paradigmas como la estructura quadtrees y las triangulaciones Delaunay y determinar su comportamiento.

También se pueden realizar estudios con diversas métricas además de la euclideana.

La metodología propuesta utiliza la identificación de pares de aristas que se intersecan en un solo punto, a través de la ecuación paramétrica de la recta, sin embargo, se puede implementar un metodología que identifique otros tipos de intersecciones, que permita diseñar estrategias distintas para descruzar cada tipo de cruce, haciendo más eficiente el número de pasos para lograr un circuito hamiltoniano libre de cruces. A partir de los resultados obtenidos en cuanto al tiempo, se puede conjeturar empíricamente que el espacio de búsqueda se reduce al 12% en promedio aproximadamente, esto se obtiene tomando el

porcentaje de cruces en función del número de ciudades por cada una de las instancias, sin embargo, podría hacerse un estudio más riguroso por medio de un análisis teórico para determinar con exactitud el tamaño de vecindad a partir de las estrategias utilizadas en este proyecto.

También puedo concluir que el atractivo principal que me atrajo al estudio de este problema es justamente lo difícil que es encontrar una solución, por lo que resulta un desafío constante para los investigadores del área, es uno de los que más interés ha suscitado en optimización combinatoria , de hecho es uno de los "benchmarks" utilizados para evaluar nuevos métodos de optimización combinatoria la gran mayoría de las técnicas que han ido apareciendo en ésta área han sido probadas en él, puesto que su resolución es de gran complejidad y porque 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. Desde el punto de vista de ciencias de la computación teórica un problema que es muy difícil de resolver es un problema NP-Completo. Ha sido fruto de estudio incluso en el área de la psicología.

Es un problema que te invita a crear y diseñar nuevas estrategias de solución para obtener soluciones cada vez más cercanas a las óptimas.

ANEXO I

¿Qué es TSPLAB?

El TSPLAB es un programa escrito en el lenguaje de programación Visual Basic 6, esta diseñado para experimentar con cuatro algoritmos de construcción: vecino más cercano y también tiene dos algoritmos de mejora para resolver el Problema del Agente Viajero Euclidiano (PAVE) o bien el que "*vive*" en el plano, por lo que las ciudades son puntos en el plano representados mediante coordenadas geométricas "x" y "y".

El TSPLAB tiene la particularidad de que se pueden abrir instancias de la TSP LIB (Son 105 instancias manejadas a nivel mundial para poder experimentar los algoritmos propuestos de las cuales 79 tienen la distancia euclideana), éstas instancias tienen la particularidad de contar con la solución óptima del problema, por lo que es fácil determinar la eficiencia de los algoritmos propuestos , hasta el momento la instancia más grande respecto al número de nodos resuelta de manera óptima es la instancia sw24978 ciudades en el caso del TSPLAB puedes abrir todas las instancias, o bien puedes generar instancias de manera aleatoria de 10, 000 puntos o ciudades.

Instalando TSPLAB

Para poder ejecutar el software propuesto es necesario instalar el programa, para lo que se requieren dos archivos: un instalador y el programa TSPLAB. Es necesario guardar estos dos archivos en una misma carpeta para poder ejecutar el programa, se pueden tener estos dos archivos en el Escritorio. Una vez que tienes el programa listo para ejecutar, realizamos lo siguiente.

Cuando queramos emplear el software propuesto llamado TSPLAB es necesario mandar llamar el programa, con el botón Inicio-Todos los programas-TSP-TSP, como lo muestra la siguiente pantalla. O bien si se tiene un icono dispuesto podemos hacer click y mandar llamar el programa.

Para el correcto funcionamiento del programa es necesario contar con dos archivos: una carpeta que contiene todas las instancias extensión .tsl y el programa TSPLAB. Es necesario guardar estos dos archivos en una misma carpeta para poder ejecutar el programa, se pueden tener estos dos archivos en el Escritorio.

Cuando queramos emplear el software propuesto llamado TSPLAB es necesario mandar llamar el programa, con el botón Inicio-Todos los programas-TSP-TSP, como lo muestra la siguiente pantalla. O bien si se tiene un icono dispuesto podemos hacer click y mandar llamar el programa.

Funcionamiento de TSPLAB

Una vez instalado y ejecutado TSPLAB entra a su primera pantalla, mostrada en la figura 1.

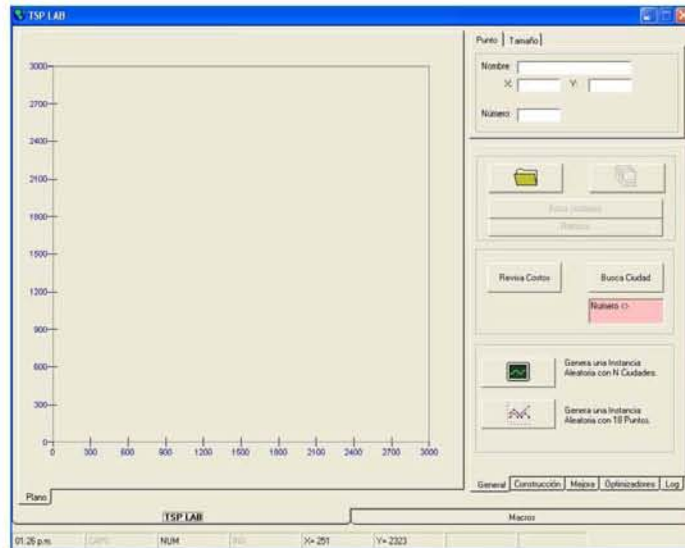


Figura 1. Pantalla principal de TSPLAB.

La pestaña General se divide en varias secciones.

En la primera sección puede modificarse algunas características como el tamaño del punto para hacerlo más visible dentro del plano e identificar sus coordenadas y que cuidad que representa.

En la figura 2 se muestra la pestaña que nos brinda información acerca de los puntos en el plano, esto se logra colocar el cursor sobre algún punto; en la pestaña despliega que ciudad representa, las coordenadas y su número dentro de la instancia o modelo.

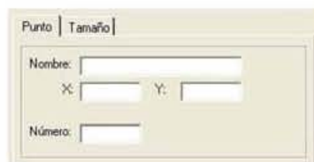


Figura 2. Pestaña que indica tamaño del punto y ubicación con coordenadas *x* e *y*

En la figura 3 se muestra la pestaña de tamaño, nos permite cambiar el tamaño de los puntos en el plano para hacerlos mas identificables, se mueve la barra deslizable a la derecha y se hace clic en cambia tamaño.

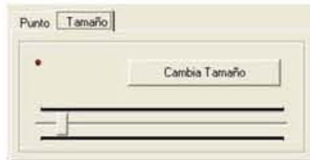


Figura 3. Pantalla de cambio de tamaño de punto

En la segunda sección encontramos algunas aplicaciones típicas del entorno Windows, como son abrir y guardar Figura 4.

Figura 4. Iconos de guardar y abrir archivos

Abre archivo de ciudades te remite de manera inmediata a la carpeta donde se han almacenado las instancias (archivos .tsl). Si se ha trabajado previamente con otra instancia antes despliega un mensaje (figura 5).

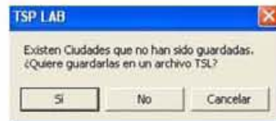


Figura 5. Mensaje para guardar instancia generada por el usuario

Guardar, permite al usuario guardar o modificar una instancia y despliega el mensaje de la figura 6.

Figura 6. Icono de sobrescribir la instancia generada

Borra ciudades, cierra la instancia con la que se este trabajando.

Reinicia, permite limpiar todas las operaciones hechas a una instancia y la deja en su estado original para aplicar otros algoritmos. Hace que permanezcan los puntos.

En la tercera sección hay dos botones, revisa costos y busca ciudad. Figura 7.

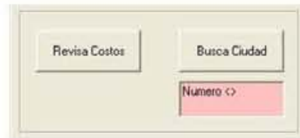


Figura 7. Iconos de revisa costos y buscar ciudades.

Al oprimir el botón es revisa costos se despliega la matriz de costos, que despliega un archivo Excel con el costo del tour entre ciudades. Figura 8.

TSP LAB

	C 1	C 2	C 3	C 4	C 5	C 6	C 7	C 8	C 9	C 10	C 11	C 12
C 1	0.0	6365.4	1702.7	8650.7	6264.6	2438.0	3132.5	5491.7	4851.8	7318.2	4317.8	3932.2
C 2	6365.4	0.0	4662.7	2300.8	1033.3	5817.3	5542.1	5123.7	4678.0	1136.0	3256.4	3810.1
C 3	1702.7	4662.7	0.0	6949.3	4592.8	2338.2	2732.1	4601.6	3893.3	5624.6	2924.7	2696.4
C 4	8650.7	2300.8	6949.3	0.0	2798.7	8063.7	7726.8	6877.3	6580.3	1401.7	5365.6	5946.0
C 5	6264.6	1033.3	4592.8	2798.7	0.0	5329.5	4945.9	4199.9	3823.6	2011.2	2572.6	3156.1
C 6	2438.0	5817.3	2338.2	8063.7	5329.5	0.0	742.8	3208.4	2669.2	6928.3	2830.1	2265.7
C 7	3132.5	5542.1	2732.1	7726.8	4945.9	742.8	0.0	2467.3	1951.5	6672.6	2379.7	1794.4
C 8	5491.7	5123.7	4601.6	6877.3	4199.9	3208.4	2467.3	0.0	717.9	6202.4	2240.4	2050.4
C 9	4851.8	4678.0	3893.3	6580.3	3823.6	2669.2	1951.5	717.9	0.0	5788.3	1601.5	1342.5
C 10	7318.2	1136.0	5624.6	1401.7	2011.2	6928.3	6672.6	6202.4	5788.3	0.0	4391.5	4946.1
C 11	4317.8	3256.4	2924.7	5365.6	2572.6	2830.1	2379.7	2240.4	1601.5	4391.5	0.0	585.4
C 12	3932.2	3810.1	2696.4	5946.0	3156.1	2265.7	1794.4	2050.4	1342.5	4946.1	585.4	0.0
C 13	4538.3	2512.0	2972.5	4679.0	1923.1	3406.5	3050.5	2919.8	2329.7	3647.9	765.0	1298.3
C 14	5155.9	2387.2	3613.6	4377.0	1579.2	3853.2	3404.9	2761.6	2290.9	3500.0	1029.0	1611.9
C 15	4069.4	4138.5	2961.9	6224.8	3426.3	2177.3	1603.2	1686.7	969.7	5273.3	882.4	405.3
C 16	6097.3	4177.8	4895.1	5708.6	3178.2	4075.5	3381.3	1303.8	1450.6	5182.1	2039.7	2207.2
C 17	1780.9	6134.4	2109.6	8417.0	5748.3	726.0	1468.5	3931.3	3375.2	7215.6	3352.4	2823.4
C 18	3227.2	5402.2	2737.9	7577.3	4792.8	880.2	167.9	2330.3	1795.9	6534.3	2223.6	1638.2
C 19	2245.0	6036.5	2345.2	8295.4	5577.0	292.6	1019.5	3486.4	2958.0	7139.7	3099.6	2541.8
C 20	3280.8	3900.1	2003.5	6134.9	3408.2	1929.2	1680.2	2668.5	1950.6	5021.1	1048.2	693.7

Figura 8. Matriz de costos entre ciudades

¿Qué es la matriz de costos?

La matriz de costos es el costo asociado de viajar de una ciudad a otra, es una matriz completa nxn y tiene la particularidad de formar una matriz triangular superior cuyos elementos de la diagonal principal (es decir el elemento cii es igual a cero o a infinito).

Busca ciudad permite buscar una ciudad escribiendo su nombre en el campo en color rosa; podemos también escribir el número de la ciudad y después oprimir el botón y así encontrarla mediante el parpadeo del punto que representa la ciudad en el plano. Figura 9.

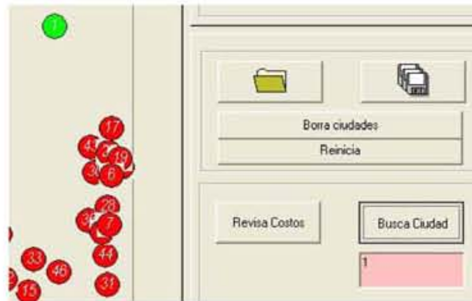


Figura 9. Localización de la ciudad 1.

En la cuarta sección encontramos dos botones, uno de ellos genera una instancia aleatoria con n ciudades, donde n pude ser desde una ciudad hasta 10000. Figura 10.

Figura 10. Icono para generar instancias aleatorias de n o 18 ciudades

El primer botón abre el siguiente mensaje donde se pide el número de nodos para generar una nueva instancia aleatoria. Figura 11.

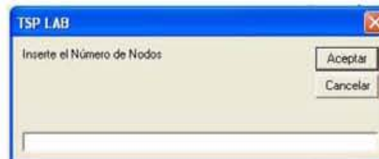


Figura 11. Icono que se abre cuando se selecciona la opción genera instancia aleatoria

Y tenemos un segundo botón que genera una instancia aleatoria con 18 puntos el cual resuelve una aplicación inherente a este software que es el test de Hartmann. Al cual pueden asignarse valores como se muestra en la siguiente figura:

TSP LAB

	Valor X	Valor Y
Punto 1	0	0
Punto 2	2	16
Punto 3	6	1
Punto 4	6	8
Punto 5	6	19
Punto 6	20	8
Punto 7	6	3
Punto 8	3	13
Punto 9	8	8
Punto 10	14	7
Punto 11	13	4
Punto 12	4	12
Punto 13	2	9
Punto 14	18	5
Punto 15	16	8
Punto 16	6	18
Punto 17	13	13
Punto 18	9	2

Genera Instancia Hartman

Figura 11. Una solución al test de Hartman.

La pestaña construcción

Algoritmos de construcción.

El primer algoritmo de construcción que se tiene es el de vecino cercano, es un algoritmo iterativo el cual consiste de ir añadiendo en cada uno de los pasos una ciudad, se comienza con un numero arbitrario; en este caso el punto número uno o ciudad número uno y va seleccionando de manera iterativa el punto más cercano hasta completar el circuito hamiltoniano. Figura 12

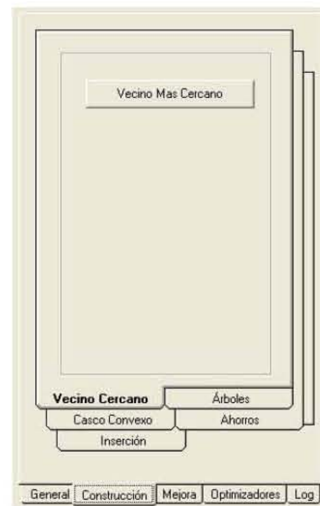


Figura 12. Botón del algoritmo Vecino Más Cercano.

Otra posibilidad que tenemos de construir el circuito hamiltoniano es mediante árboles , se tienes cuatro posibilidades y botones el primero de ellos es prim el cual construye un árbol de expansión de mínimo peso, de hecho la descripción ya esta en la tesis al igual que kruscal, después tenemos los algoritmos que generan a partir de un árbol de expansión un circuito hamiltoniano , el primero de ellos es el de duplicación el cual entrega un circuito hamiltoniano cuyo costo es a lo más el doble del costo optimo y christofides el cual se basa en algún algoritmo de construcción del árbol ya sea prim o kruscal y te permite también encontrar un circuito hamiltoniano cuyo costo esta alejado en 3/2 del optimo. Figura 13.

Figura 13. Pestaña que muestra los algoritmos para construir árboles

La siguiente pestaña se incluyen los algoritmos de inserción, son básicamente cuatro, inserción más cercana, mas alejada, inserción aleatoria e inserción más barata, cada uno de ellos despliega una posibilidad de construir un circuito hamiltoniano de acuerdo a un criterio de inserción. Figura 14.

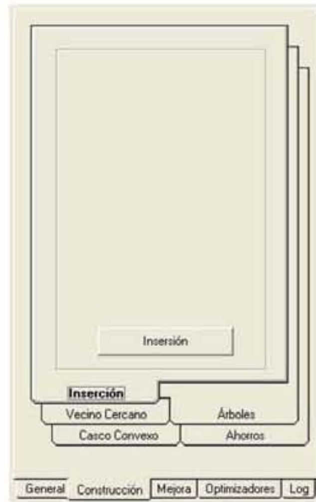


Figura 14. Botones de Algoritmos de Inserción.

Después tenemos los de casco convexo, que construyen un circuito hamiltoniano con base en la estructura geométrica de casco convexo, el cual consiste primero en la construcción del casco convexo e ir llamando de una manera iterativa hasta tener el circuito hamiltoniano.

Figura 15. Icono del algoritmo de casco convexo.

Y por ultimo tenemos los algoritmos basados en ahorros en los cuales se parte de la idea construir circuitos hamiltonianos a partir de sub-circuitos. Figura 16.

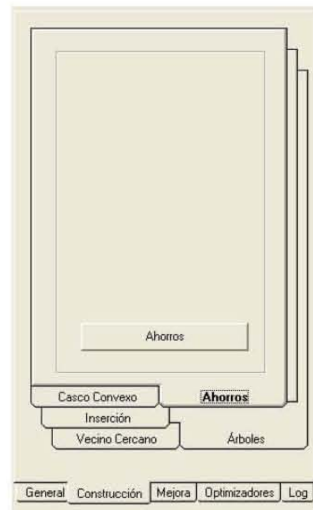


Figura 16. Icono del algoritmo de ahorros.

La Pestaña mejora.

Tenemos la posibilidad de buscar y encontrar cruces en el circuito hamiltoniano obtenido en el paso anterior, el algoritmo actúa en un tiempo polinomial y lo que hace es detectar cruces mas no te dice que punto de intersección tiene, este busca cruces de tal forma que el segmento se parametriza y se comparan todos contra todos y si el determinante en cualquier par de los segmentos es diferente de cero indica que si hay posibilidad de cruce y si los parámetros s y t son diferentes de cero, quiere decir que los segmentos se están intersectando. Figura 17

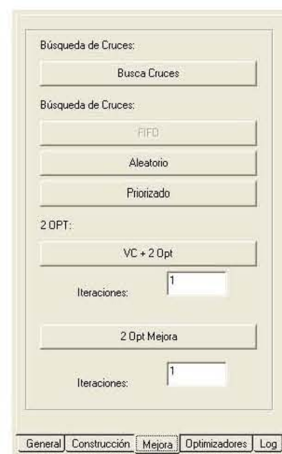


Figura 17. Pestaña de algoritmos de Mejora.

Toda esta información se agrega al reporte que se genera en la pestaña log

Figura 18. Botón algoritmo busca cruces con tres criteros

En el botón FIFO (first in first out), es decir, como ya tengo detectados los segmentos que se están intersecando, de esa lista de segmentos se toman los dos primeros y se descruzan, después de manera aleatoria o de manera priorizada. Descruza pares de listas cruzadas.

Este algoritmo lo que hace es priorizar que segmento vas a descruzar, en este caso el segmento que se toma primero es aquel cuya longitud sea el mas largo de todos, de toda la lista de segmentos que se cruzan arrojado por busca cruces me tomo el que tenga mayor longitud y lo descruzo.

Después la técnica de 2opt ejecuta el algoritmo sin tomar en cuenta la lista que se genera de pares de aristas que se intersecan. Intercambia los pares de aristas de manera exhaustiva encontrando el par que arroje la mejor solición. La ventana o campo me permite indicar cuantas iteraciones quieres. De manera experimental se puede observar el número de iteraciones que se requieren para alcanzar un circuito libre de cruces.

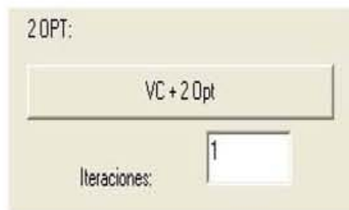


Figura 19. Botón algoritmo 2-Opt

Después tenemos otro botón que ejecuta el algoritmo de mejora propuesto, basado en el algoritmo anterior, en este caso 2opt Mejora lo que hace es tomar una lista previa, es decir le quitamos la miopía y únicamente intercambia los pares de segmentos de la lista L.

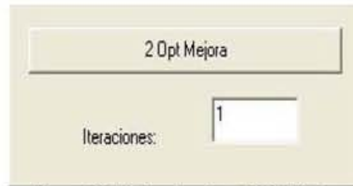


Figura 20. Botón algoritmo 2-Opt Mejora

La pestaña optimizadores

Se refiere al conjunto de algoritmos que permitirán identificar el casco convexo de cualquier instancia una vez que se ha ejecutado algun algoritmo de construcción. Es decir, puedes generar un casco convexo, figura 21 o bien quadtrees según el subconjunto de ciudades que requieras.

El casco convexo garantiza que el área formada por estos puntos es un área mínima y que los puntos extremos son los puntos frontera. Todos los demás puntos están contenidos en esta envolvente convexa.

Figura 22. Boton que genera un casco convexo

Figura 23. Boton que genera quadtrees

El icono de generar quadtrees, el cual te permite ejecutar una estructura geométrica que es una partición recursiva del plano alineados a los ejes x,y y te permite hacer una partición recursiva con el numero de ciudades que tu puedes elegir mediante el umbral, este umbral es el número de ciudades que quieres que queden encasilladas en un cuadro y es previo al genera quadtrees.

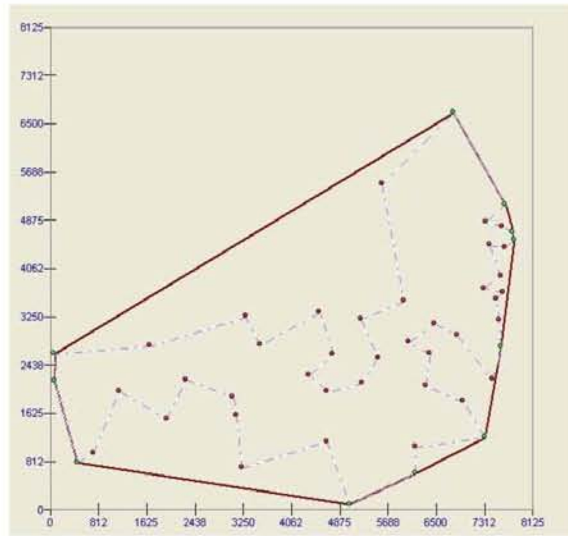


Figura 24. Ilustración de casco convexo.

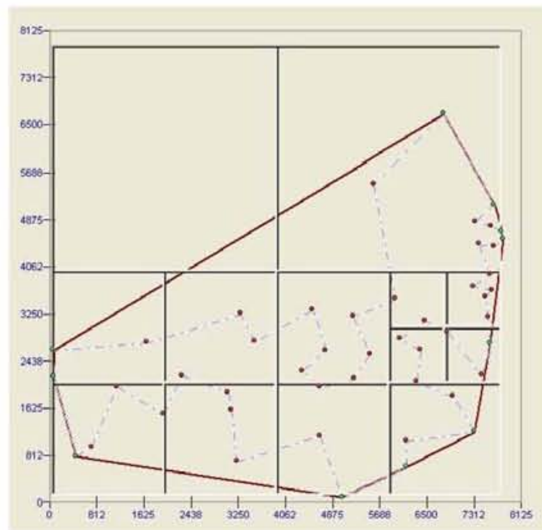


Figura 25. Ilustración de Quadtrees

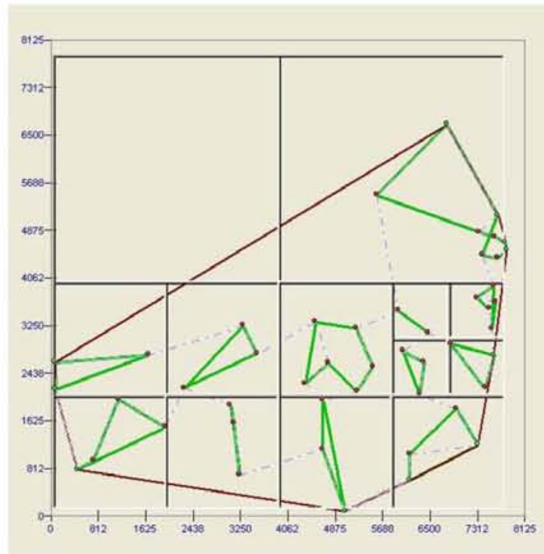


Figura 26. Generación de Tour

Para que puedas ejecutar genera quadtrees previamente se puede seleccionar el numero de ciudades encasilladas por un cuadro, este umbral va desde una cuidad agrupada hasta n ciudades, donde n es el número de ciudades agregadas de la instancia, en este caso podemos ponerle una ejecución de 5 ciudades y entonces en cada cuadro puede haber desde una, dos , tres, cuatro, hasta 5 ciudades.

Después al dar clic en el boton construye tours , lo que hace es construir los tours entre ese numero de ciudades. Figura 26.

Una vez que ejecutamos el optimizador quadtrees se despliega de manera paralela una pestaña llamada reporte, este guarda información de cada uno de los cuadrantes, Una partición que indica como estan contenidas y relacionadas las ciudades, de acuerdo con el optimizador. Figura 27

Figura 27. Icono de Reporte

Podemos dar clic en cada uno de los signos mas (+) para que despliegue las ciudades contenidas en cada uno de los cuadrantes. Despliega la información de los cuadrantes y su tour, del lado derecho esta para expandir o contraer el árbol, o ir paso a paso, si nos colocamos sobre una ciudad la identifica mencionando nombre , sus coordenadas x y,, y si esta palomeada pertenece al casco convexo. Figura 28.

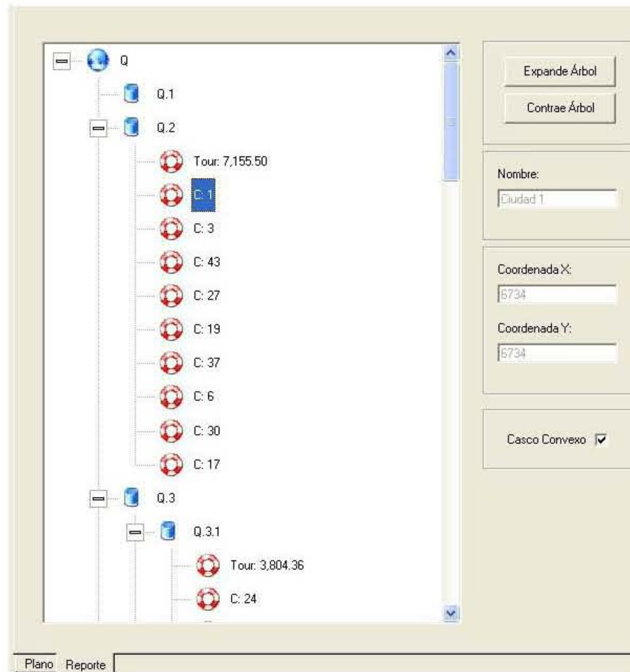


Figura 28. Reporte arrojado por el algoritmo quadtree

La pestaña log

Toda la ejecución de los algoritmos se guarda forma de resumen en una pantalla llamada log. Te permite ver todos los procedimientos que han sido ejecutados desde el nombre de la instancia y hasta el nombre del algoritmo ejecutado. El tour generado con todos los números de ciudades o ya sea con los dos algoritmo de construcción o el mejorado; y tienes la posibilidad de borrar ese historial porque almacena toda la sesión de ejecución de todas las instancias del programa hasta que se cierra. Figura 29

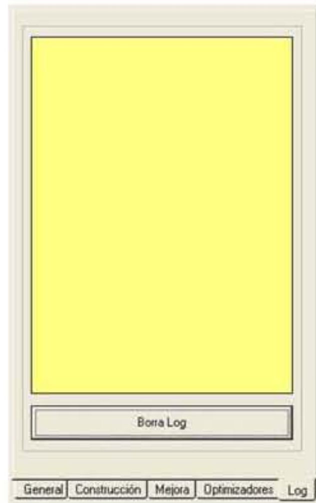


Figura 29.Pantalla que muestra todo el historial de instrucciones ejecutadas

La pestaña macros

Lo que te permite hacer las macros es trabajar de manera simultánea los diferentes algoritmos, con una instancia o diferentes instancias, de tal manera que en la macro 1 tienes la posibilidad de seleccionar un archivo dando clic en el botón " archivoTSL " el cual mandara llamar el archivo donde se encuentra localizadas las instancias que son las previamente diseñadas o que tu mismo las has generado. Seleccionando la instancia con la que quieras trabajar. Figuras 29 y 30.

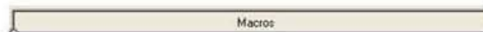


Figura 29.Icono de Macros

En los algoritmos de construcción tiene un menú colgante que tiene diferentes opciones, puede ser vecino más cercano, duplicación, kristofides, ahorros o algún algoritmo de inserción. En el campo del lado derecho tienes la posibilidad de poner el número de iteraciones que quieres que corra.

Figura 30.Ventana con campos para seleccionar diversos algoritmos para ejecutarse en un solo tiempo

Tenemos otro menú colgante donde se incluyen las técnicas de mejora que son 2 OPT o 2 OPTMejorada y también tiene el campo para el número de iteraciones deseadas. Figura 30.

Figura 30. Ejemplo de menú colgante.

También podemos correrlo con un optimizador de Quad Trees o casco convexo y también tenemos el campo para indicar el número de veces que deseamos corra este algoritmo.

Finalmente, una vez llenados todos los campos podemos poner el número total de iteraciones en que se resuelve, es decir una vez llenado los campo de la macro uno, podemos utilizar más macros cambiando de instancia y cambiando de algoritmos de construcción, mejora u optimizador y así en cada una hasta 5 posibilidades. Puedes entonces correr la misma instancia con diferente macro o bien puedes correr diferentes instancias con diferentes algoritmos, para correr los algoritmos se oprime el botón ejecutar.

Referencias

[1] Lawler, E. The traveling salesman problem: a guide tour of combinatorial optimization, 1a edición, Gran Bretaña, Jhon Wiley, 1985.

[2] G. Dantzig, R. Fulkerson, and S. Johnson, "Solution of a large-scale traveling-salesman problem", Operations Research 2, 393-410. 1954.

[3] 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.

[4] A. Lodi, A. Punnen (2002).TSP Software.In G. Gutin, A. Punnen (Eds.).The Traveling Salesman Problems and its Variations, Kluwer Academic Publishers, *Combinatorial Optimization*, Dordrecht ,Vol.12, 2002.

[5] Segura, E. Análisis del Problema del Agente Viajero. Tesis de Maestría. DEPFI, UNAM. Ciudad Universitaria, México, D.F. 2004.

[6] M. Held y R Karp , The travelling salesman problem and minimum spanning trees: part II Math Prog., 1, pp. 6-25, 1971.

[7] D.L. Miller, J.F. Pekny, Exact solution of large asymmetric traveling salesman problems. Science, 251:pp. 754-761, 1991.

[8] Fiechter, C.N. A Parallel Tabu Search Algorithm for Large Scale Traveling Salesman Problems, 1990.

[9] S. Kirkpatrick, J. C. D. Gelatt, M. P. Vecchi, Optimization by Simulated Annealing, *Science*, 13 , 220(4598), pp.671-680, 1983.

[10] M. Dorigo, L. M. Gambardella , Ant colonies for the traveling salesman problem.*BioSystems*, 43:73-81, 1997.

[11] P. Larrañaga, C. M. H. Kuijpers, R. H. Murga, I. Inza, S. Dizdarevic , Genetic algorithms for the travelling salesman problem: A review of representations and operators, Articial Intelligence Review, 13, pp. 129-170, 1999.

[12] J. Cirasella, D.S. Johnson, L.A. McGeoch, W. Zhang , The asymmetric travelling salesman problem: Algorithms, instance generators, and tests. *Algorithm Engineering and Experimentation* (A.L. Buchsbaum and J. Snoeyink (editors)) *Third International Workshop*, ALENEX, 2001.

[13]Adenso, D. (Coordinador). Optimización heurística y redes neuronales en dirección de operaciones e ingeniería. Editorial:Paraninfo, Madrid, España. 1996.

[14] Johnson,L. y McGeoch. Asymptotic experimental Analysis for the Held-Karp traveling slesman bound, *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms* 1997, 341-350.

[15]K. Helsgaun, An effective implementation of the lin-kernighan travelling salesman heuristic, European Journal of Operational Research, pp.126:106-130, 2000.

[16] S. Lin and B. W. Kernighan , An effective heuristic algorithm for the Traveling Salesman problem, *Operation Research* 21 pp.498-516, 1973.

[17] Gutiérrez, A. La técnica de recocido simulado y sus aplicaciones. Tesis de Maestría. DEPFI, UNAM. Ciudad Universitaria, México, D.F. 1991.

[18] Flores, I. Apuntes de Programación Entera. Departamento de Sistemas DEPFI, UNAM, Ciudad Universitaria, México, D.F. 2002.

[19] Olivera A. Heurísticas para Problemas de Ruteo de Vehículos. Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Montevideo, Uruguay. Agosto 2004

[20] Christofides, M ."Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem,Carnegie-Mellon ,University Management sciences, *Research Report* 388, Pittsburgh, Pa.,February 1976.

[21] Arora, S. Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems, Journal of the ACM, 45(5):753-782, 1998.

[23] Garey,M. Computers and Intractability: A guide to the Theory of NP-Completeness. Freeman, New York, 1979.

[24] Marti, R. *Procedimientos Metaheuristicos en Optimización Combinatoria.* Publicaciones del departamento de Estadística e Investigación Operativa. Facultad de Matemáticas. Universidad de Valencia. 2001.

[25] Osman, I.H. y Kelly, J.P. *Meta-Heuristics: Theory and Applications*,Ed. Kluwer Academic, Boston. 1996.

[27] Jünger, M., Reinelt, G., y rinaldi, G. *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,1995,225–330.

[28] Glover, F. Ejection Chain, Reference structures and alternating path methods for traveling salesman problems, Leeds School of Business, UCB 419,University of Colorado, Boulder.1992.

[29] Ahuja R.K et al. *Estudio de técnicas de búsqueda por vecindad a muy gran escala*, Departamento de ingeniería industrial y de sistemas Universidad de Florida Gainesville,FL32611, USA, 11 octubre de 2000.

[30] Feo, T.A., et.al : *"Greedy Randomized Adaptive Search Procedures"*, Journal of Global Optimization, 6, 1995, pp.: 109–133.

[31] Dowsland K. A., et. al. *"Solving a shipper rationalisation problem with a simulated Annealing based hyperheuristic"*, The University of Nottingham, The School of Computer Science and IT, UK.

[32] Kolohan, F., Liang, M.: *"A tabu search approach to optimization of drilling operations"*, Comp. in Eng. 31, 1996, pp.: 371–374.

[33] Goldberg, D.: *"Genetic Algorithms in Search, Optimization & Machine Learning"*, Addison–Wesley, 1989.

[34] Cowling P., Kendall G. and Soubeiga E., *"A Hyperheuristic Approach to Scheduling a Sales Summit"*, In E. Burke and W. Erben (Eds.) PATAT 2000, Springer Lecture Notes in Computer Science no. 2079, 2001, pp.: 176–190.

[35] B. Chandra, H. Karloff and C. Tovey,"New results on the old k-opt algorithm for the TSP", In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 150-159 (1994).

[36] Helsagaun, Keld.,An Effective Implementation of *K*-opt Moves for the Lin-Kernighan TSP Heuristic, Computer Science , Roskilde University DK-4000 Roskilde, Denmark.

[37] E. Aarts and J.K. Lenstra, *Local Search in Combinatorial Optimization*, John Wiley & Sons, New York, 1997.

[38] Reeves, C., *Modern Heuristics Techniques for Combinatorial Problems*, Ed. McGraw-Hill.UK.1995.

[39] Osman, I.H. y Kelly, J.P. *Meta-Heuristics: Theory and Applications*,Ed. Kluwer Academic, Boston. 1996.

[40] K. Menger, "Das botenproblem", in *Ergebnisse eines Mathematischen Kolloquiums* 2 (K. Menger, editor), Teubner, Leipzig, pages 11-12. 1932.

[41] P. C. Mahalanobis, "A sample survey of the acreage under jute in Bengal", *Sankhyu* 4, 511-530. 1932

[42] Jessen, R,J., "Statistical investigation of a sample survey for obtaining farm facts", *Research Bulletin* #304, Iowa State College of Agriculture. 1945.

[43] J.B. Robinson, "On the Hamiltonian game (a traveling-salesman problem)", *RAND Research Memorandum* RM-303. 1949.

[44] G. Dantzig, R. Fulkerson, and S. Johnson, "Solution of a large-scale traveling-salesman problem", *Operations Research* 2, 393-410. 1954.

[45] I. Heller, "The travelling salesman's problem: part 1 -- basic facts", *Research Report*, George Washington University Logistics Reserch Project. 1954.

[46] I. Heller, "On the travelling salesman's problem", *Proceedings of the Second Symposium in Linear Programming (Volume 1)*, Washington, D.C., January 27-29. 1955.

[47] H.W. Kuhn, "On certain convex polyhedra", Abstract 799t, *Bulletin of the American Mathematical Society* 61, 557-558. 1955.

[48] G. Morton and A.H. Land, "A contribution to the `travelling-salesman' problem", *Journal of the Royal Statistical Society, Series B* 17, 185-194. 1955.

[49] R.Z. Norman, "On the convex polyhedra of the symmetric traveling salesman problem", Abstract 804t, *Bulletin of the American Mathematical Society* 61, 559-569. 1955.

[50] J.T. Robacker, "Some experiments on the traveling-salesman problem", *RAND Research Memorandum* RM-1521. 1955.

[51] M.M. Flood, "The traveling-salesman problem", *Operations Research* 4, 61-75. 1956.

[52] J.B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem", *Proceedings of the American Mathematical Society* 2, 48-50. 1955.

[53] L.L. Barachet, "Graphic solution of the traveling-salesman problem", *Operations Research* 5, 841-845. 1957.

[54] F. Bock, "An algorithm for solving `traveling-salesman' and related network optimization problems", *Research Report*, Armour Research Foundation. (Presented at the Operations Research Society of America Fourteenth National Meeting, St. Louis, October 24, 1958). 1958.

[55] G.A. Croes, "A method for solving traveling-salesman problems", *Operations Research* 6, 791-812. 1952.

[56] W.L. Eastman, "Linear programming with pattern constraints", Ph.D. Dissertation, Harvard. 1958.

[57] M.J. Rossman and R.J. Twery, "A solution to the travelling salesman problem", *Operations Research* 6, page 687, Abstract E3.1.3. 1958.

[58] G.B. Dantzig, D.R. Fulkerson, and S.M. Johnson, "On a linear-programming, combinatorial approach to the traveling-salesman problem", *Operations Research* 7, 59-66. 1959.

[59] W. Riley, III, "A new approach to the traveling-salesman problem", *Operations Research (Supplement 1)* 7, Abstract B4. 1959.

[60] R. Bellman, "Combinatorial processes and dynamic programming", in: *Combinatorial Analysis* (R. Bellman and M. Hall, Jr., eds.), American Mathematical Society, pp. 217-249. 1960.

[61] M.F. Dacey, "Selection of an initial solution for the traveling-salesman problem", *Operations Research* 8, 133-134. 1960.

[62] F. Lambert, "The traveling-salesman problem", *Cahiers du Centre de Recherche Opérationelle* 2, 180-191. 1960

[63] C.E. Miller, A.W. Tucker, and R.A. Zemlin, "Integer programming formulation of traveling salesman problems", *Journal of the Association for Computing Machinery* 7, 326-329. 1960.

[64] W. Riley, III, "Micro-analysis applied to the travelling salesman problem", *Operations Research (Supplement 1)* 8, Abstract D4. 1960.

[65] E.L. Arnoff and S.S. Sengupta, "Mathematical programming", in: *Progress in Operations Research* (R.L. Ackoff, editor), John Wiley and Sons, New York, pp. 105-210. 1961.

[66] H. Müuller - Merbach, "Die ermittlung des kürzesten rundreiseweges mittels linear programmierung", *Ablauf und Planungsforschung* 2, 70-83. 1961.

[67] B. Thüring, "Zum problem der exakten ermittlung des kürzesten rundreiseweges", *Elektronische Datenverarbeitung* 3, 147-156. 1961.

[68] R. Bellman, "Dynamic programming treatment of the travelling salesman problem", *Journal of the Association for Computing Machinery* 9, 61-63. 1962.

[69] R.H. Gonzales, "Solution to the traveling salesman problem by dynamic programming on the hypercube", *Technical Report* Number 18, Operations Research Center, Massachusetts Institute of Technology. 1962.

[70] M. Held and R.M. Karp, "A dynamic programming approach to sequencing problems", *Journal of the Society of Industrial and Applied Mathematics* 10, 196-210. 1962.

[71] F. Bock, "Mathematical programming solution of traveling salesman examples", in: *Recent Advances in Mathematical Programming* (R.L. graves and P. Wolfe, eds.), McGraw-Hill, New York. 1963.

[72] J.D.C. Little, K.G. Murty, D.W. Sweeney, and C. Karel, "An algorithm for the traveling salesman problem", *Operations Research* 11, 972-989. 1963.

[73] V.I. Mudrov, "A method of solution of the traveling salesman problem by means of integer linear programming (the problem of finding the Hamiltonian paths of shortest length in a complete graph)" (in Russian), *Zhurnal Vychislennoi Fiziki (USSR)* 3, 1137-1139. (Abstract in: *International Abstracts in Operations Research* 5 (1965), Abstract Number 3330.) 1963.

[74] R.L. Karg and G.L. Thompson, "A heuristic approach to solving travelling salesman problems", *Management Science* 10, 225-248. 1964.

[75] W. Suurballe, "Network algorithm for the traveling salesman problem", *Operations Research (Supplement 1)* 12, Abstract 3Tc.1. 1964.

[76] S. Lin, "Computer solutions of the traveling salesman problem", *The Bell System Technical Journal,* No. 10, Diciembre , pp. 2245-2269, 1965.

[77] S. Reiter and G. Sherman, "Discrete optimizing", *SIAM Journal on Applied Mathematics* 13, Septiembre de 1965, pp. 864-889.

[78] N.P. Salz, "The NORMA: a possible basis for a theory for the traveling-salesman problem", *Operations Research (Supplement 1)* 13, Abstract C4h. 1965.

[79] R.E. Gomory, "The traveling salesman problem", in: *Proceedings of the IBM Scientific Computing Symposium on Combinatorial Problems*, IBM, White Plains, pp. 93-121. 1966.

[80] E.L. Lawler and D.E. Wood, "Branch-and-bound methods: a survey", *Operations Research* 14, 699-719. 1966.

[81] G.T. Martin, "Solving the traveling salesman problem by integer linear programming", *Operations Research (Supplement 1)* 14, Abstract WA7.10. 1966.

[82] G.T. Martin, "Solving the traveling salesman problem by integer linear programming", C-E-I-R, New York, 1966.

[83] H. Mueller-Merbach, "Drei neue methoden zur lösung des travelling salesman problems, teil 1", *Ablauf und Planungsforschung* 7, 32-46. 1966.

[84] H. Mueller-Merbach, "Drei neue methoden zur lösung des travelling salesman problems, teil 2", *Ablauf und Planungsforschung* 7, 78-91. 1966.

[85] S.M. Roberts and B. Flores, "An engineering approach to the traveling salesman problem", *Management Science* 13, 269-288. 1966.

[86] N.P. Salz, "The NORMA: a theory for the traveling salesman problem", *Operations Research (Supplement 2)* 14, Abstract MP3.11. 1966.

[87] D. Shapiro, "Algorithms for the solution of the optimal cost traveling salesman problem", Sc.D. Thesis, Washington University, St. Louis. 1966.

[88] B. Fleischmann, "Computational experience with the algorithm of Balas", *Operations Research* 15, 153-154. 1967.

[89] M. Bellmore and G.L. Nemhauser, "The traveling salesman problem: a survey", *Operations Research* 16, 538-558. 1968.

[90] P. Pfluger, "Diskussion der modellwahl am beispiel des traveling-salesman problems", in: *Einführung in die Methode Branch and Bound* (M. Bechmann and H.P. Künzi, eds.), *Lecture Notes in Operations Research and Mathematical Economics* 4, Springer, Berlin, pp. 88-106. 1968.

[91] A.M. Issac and E. Turban, "Some comments on the traveling salesman problem", *Operations Research* 17, 543-546. 1969.

[92] T.C. Raymond, "Heuristic algorithm for the traveling-salesman problem", *IBM Journal of Research and Development* 13, 400-407. 1969.

[93] M. Held and R.M. Karp, "The traveling-salesman problem and minimum spanning trees", *Operations Research* 18, 1138-1162. 1970.

[94] H. Müller-Merbach, *Optimale Reihenfolgen*, Springer, Berlin. 1970.

[95] M. Bellmore and J. Malone, "Pathology of traveling-salesman subtour-elimination algorithms", *Operations Research* 19, 278-307. 1971.

[96] M. Held and R.M. Karp, "The traveling-salesman problem and minimum spanning trees: part II", *Mathematical Programming* 1, 6-25. 1971.

[97] P. Krolak, W. Felts, and G. Marble, "A man-machine approach toward solving the traveling salesman problem", *Communications of the ACM* 14, 327-334. 1971.

[98] N. Christofides, "Bounds for the travelling-salesman problem", *Operations Research* 20, 1044-1056. 1972.

[99] S. Hong, "A linear programming approach for the traveling salesman problem", Ph.D. Thesis, The Johns Hopkins University. 1972.

[100] H. Steckhan and R. Thome, "Vereinfachungen der Eastmanischeen branch-bound-lösung für symmetrische traveling salesman probleme", *Methods of Operations Research* 14, 360-389. 1972.

[102] K. Helbig Hansen and J. Krarup, "Improvements of the Held-Karp algorithm for the symmetric traveling-salesman problem", *Mathematical Programming*7, 87-96. 1974.

[103] M. Held, P. Wolfe, and H.P. Crowder, "Validation of subgradient optimization", *Mathematical Programming* 6, 62-88. 1974. 1974.

[104] P.M. Camerini, L. Fratta, and F. Maffioli, "On improving relaxation methods by modified gradient techniques", *Mathematical Programming Study*3, 26-34. 1974.

[105]W. Schiebel, G. Unger, and J. Terno, "A cutting plane algorithm for the travelling salesman problem", *Report* 07-15-76, Sektion Mathematik, Technische Universität Dresden, GDR. 1975.

[106]P. Miliotis, "Integer programming approaches to the travelling salesman problem", *Mathematical Programming* 10, 367-378. 1976.

[107]M.S. Bazaraa and J.J. Goode, "The traveling salesman problem: a duality approach", *Mathematical Programming* 13, 221-237. 1977.

[108]M. Grötschel, *Polyedrische Charackterisierungen kombinatorischer Optimierungsprobleme*, Anton Hain Verlag, Meisenheim/Glan. 1977.

[109]T.H.C. Smith and G.L. Thompson, "A LIFO implicit enumeration search algorithm for the symmetric traveling salesman problem using Held and Karp's 1-tree relaxation", in: *Studies in Integer Programming* (P.L. Hammer, E.L. Johnson, B.H. Korte, and G.L. Nemhauser, eds.), *Annals of Discrete Mathematics*1, North-Holland, Amsterdam, pp. 479-493. 1977.

[110]M. Grötschel and M. Padberg, "On the symmetric traveling salesman problem: theory and computation", in: *Optimization and Operations Research* (R. Henn, B. Korte, and W. Oettli, eds.), *Lecture Notes in Economics and Mathematical Systems* 157, Springer, Berlin, pp. 105-115. 1978. 1978.

[111]P. Miliotis, "Using cutting planes to solve the symmetric travelling salesman problem", *Mathematical Programming* 15, 177-188. 1978.

[112]R.E. Burkard, "Travelling salesman and assignment problems: a survey", in: *Discrete Optimization 1* (P.L. Hammer, E.L. Johnson, and B.H. Korte, eds.), *Annals of Discrete Mathematics* 4, North-Holland, Amsterdam, pp. 193-215. 1979.

[113]A. Land, "The solution of some 100-city travelling salesman problems", *Technical Report, London School of Economics*. 1979.

[114]H. Crowder and M.W. Padberg, "Solving large-scale symmetric travelling salesman problems to optimality", *Management Science* 26, 495-509. 1980.

[115]M. Grötschel, "On the symmetric travelling salesman problem: solution of a 120-city problem", *Mathematical Programming Study* 12, 61-77. 1980.

[116]D.J. Houck, Jr., J.C. Picard, M. Queyranne, and R.R. Vemuganti, "The travelling salesman problem as a constrained shortest path problem: theory and computational experience", *OPSEARCH* 17, 93-109. 1980.

[117]M.W. Padberg and S. Hong, "On the symmetric travelling salesman problem: a computational study", *Mathematical Programming Study* 12, 78-107. 1980.

[118]J. Mohan, "A study in parallel computation - the traveling salesman problem", Technical *Report* CMU-CS-82-136, *Computer Science Department, Carnegie Mellon University*, 1982.

[119]T. Volgenant and R. Jonker, "A branch and bound algorithm for the symmetric traveling salesman problem based on the 1-tree relaxation", *European Journal of Operational Research* 9, 83-89. 1982.

[120]B. Gavish and K.N. Srikanth, "Algorithms for solving large-scale symmetric traveling salesman problems to optimality", *Working Paper Series* Number QM8329, Graduate School of Management, University of Rochester. 1983.

[121]T. Volgenant and R. Jonker, "The symmetric traveling salesman problem and edge exchanges in minimal 1-trees", *European Journal of Operational Research* 12, 394-403. 1983.

[122]R. Jonker and T. Volgenant, "Nonoptimal edges for the symmetric traveling salesman problem", *Operations Research* 32, 837-846. 1984.

[123]B. Fleischmann, "A cutting plane procedure for the travelling salesman problem on road networks", *European Journal of Operational Research* 21, 307-317. 1985.

[124]O.A. Holland, "Schnittebenenverfahren für travelling-salesman und verwandte probleme", Dotoral Thesis, Universität Bonn. 1987.

[125]M. Grötschel and O. Holland, "A cutting plane algorithm for minimum perfect 2-matchings", *Computing* 39, 327-344. 1987.

[126]M. Padberg and G. Rinaldi, "Optimization of a 532-city symmetric traveling salesman problem by branch and cut", *Operations Research Letters*6, 1-7. 1987.

[127]G. Carpaneto, M. Fischetti, and P. Toth, "New lower bounds for the symmetric travelling salesman problem", *Mathematical Programming* 45, 223-254. 1988.

[128]J. Rost and E. Maehle, "Implementation of a parallel branch-and-bound algorithm for the travelling salesman problem", in: *CONPAR 88: Proceedings*, (C.R. Jesshop and K.D. Reimartz, eds.), *The British Computer Society Workshop Series*, Cambridge University Press, New York, pp. 152-159. 1989.

[129]T.H.C. Smith, T.W.S. Meyer, "Lower bounds for the symmetric travelling salesman problem from lagrangean relaxations", *Discrete Applied Mathematics* 26, 209-217. 1990.

[130]T. Volgenant and R. Jonker, "Fictitious upper bounds in an algorithm for the symmetric traveling salesman problem", *Computers and Operations Research* 17, 113-117. 1990.

[131]M. Padberg and G. Rinaldi, "An efficient algorithm for the minimum capacity cut problem", *Mathematical Programming* 47, 19-36. 1990.

[132]M. Padberg and G. Rinaldi, "Facet identification for the symmetric traveling salesman polytope", *Mathematical Programming* 47, 219-257. 1990.

[133]M. Grötschel and O. Holland, "Solution of large-scale symmetric travelling salesman problems", *Mathematical Programming* 51, 141-202. 1991.

[134]M. Padberg and G. Rinaldi, "A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems", *SIAM Review* 33, 60-100. 1991.

[135]S. Tschöke, M. Räcke, R. Lüling, and B. Monien, "Solving the traveling salesman problem with a parallel branch-and-bound algorithm on a 1024 processor network", *Technical Report*, Department of Mathematics and Computer Science, University of Paderborn, Germany, 1992.

[136]J.-M. Clochard and D. Naddef, "Using path inequalities in a branch and cut code for the symmetric traveling salesman problem", in *Third IPCO Conference*, (G. Rinaldi and L. Wolsey, eds), pp. 291-311. 1993.

[137]M. Jünger, S. Thienel, and G. Reinelt, "Provably good solutions for the traveling salesman problem", *Zeitschrift für Operations Research* 40, 183-217. 1994.

[138]D. Applegate, R. Bixby, V. Chvátal, and W. Cook, "Finding cuts in the TSP (A preliminary report)", *DIMACS Technical Report* 95-05, March. 1995.

[139]T. Christof and G. Reinelt, "Parallel cutting plane generation for the TSP - Extended Abstract", *Research Report*, Interdisziplinäres Zentrum für Wissenschaftliches Rechnen der Universität Heidelberg. 1995.

[140]M. Jünger, G. Reinelt, and G. Rinaldi, "The traveling salemsan problem", in: *Handbooks in Operations Reseach and Management Science, Volume 7* (M.O. Ball, T. Magnanti, C.L. Monma, and G. Nemhauser, eds), Elsevier Science B.V., pp. 225-330. 1995.

[141]M. Jünger and P. Stömer, "Solving large-scale traveling salesman problems with parallel branch-and-cut", *Report* Number 95.191, Institut für Informatik, Universität Köln. 1995.

[142]S. Tschöke, R. Lüling, and B. Monien, "Solving the traveling salesman problem with a parallel branch-and-bound algorithm on a 1024 processor network", *Technical Report Number* 160, Department of Mathematics and Computer Science, University of Paderborn, Germany. 1995.

[143] R.M. Karp. Reducibility among combinatorial problems. In: R.E. Miller and J.W. Thatcher (Eds.), Complexity of computer computations, Plenum Press, New York, 1972, pp. 85-103.
[144]Papadimitriou, C.H. The Euclidean traveling salesman problema is NP-Complete. Theorical Computer Science, 4(3):237-244,1977.

[145] R. Bellman, Dynamic Programming Treatment of the Travelling Salesman Problem, Journal Journal of the ACM (JACM) Volume 9 Issue 1, Jan. 1962

[146] Bellman, R. "Combinatorial Processes and Dynamic Programming", in Bellman, R., Hall, M., Jr. (eds.), *Combinatorial Analysis, Proceedings of Symposia in Applied Mathematics 10,*, American Mathematical Society, pp. 217–249, 1960.

[147] Karp, R.M. "Dynamic programming meets the principle of inclusion and exclusion", Oper. Res. Lett. 1: 49–51,1982.

[148] Deineko V, Hoffman M and Okamoto Y., Woeginger G. The traveling salesman problem with few inner points. Proc. 10th COCOON, Lect. Notes Comput. Sci. , n.p., 2004. 268-277.

[149] Rafael Martí, Procedimientos Metaheurísticos en Optimización Combinatoria, Matemàtiques 1(1), 3-62, 2003.

[150] Thomas Stützle .The Traveling Salesman Problem: State of the Art. Workshop on Vehicle Routing, July 10, 2003.

[151] Sahni, S. and T. Gonzalez. Pcomplete Aproximation Problems. Journal of the ACM 23, 555565,1976.

[152] M. Goemans. Worst-case comparison of valid inequalities for the TSP. Mathematical Programming, 69: 335–349, 1995.

[153] R. M. Karp. Probabilistic analysis of partitioning algorithms for the TSP in the plane. *Math.Oper. Res.* **2**:209-224, 1977.

[154]J. Mitchell. Guillotine subdivisions approximate polygonal subdivisions: Part II- A simple PTAS for geometric *k*-MST, TSP, and related problems. Preliminary manuscript, April 30, 1996. To appear in *SIAM J. Computing.*

[155] Bárbara Rodeker, M. Virginia Cifuentes and Liliana Favre. An Empirical Analysis of Approximation Algorithms for Euclidean TSP. Documento PDF.

[156] Flood, M.M. The Traveling Salesman problema. Operations Research, 4, 61-75. 1956.

[157]J.L. Bentley, *Experiments on Geometric Traveling Salesmn Heuristics*, Tech Report, AT&T Bell Laboratories, Murray Hill, 1990.

[158]M. Englert, H. Röglin y B. Vöcking, *Worst case and probabilistic analysis of the 2-OPT algorithm for the TSP*, Proc. of the 18th ACM-SIAM Symposium on Discrete Algorithms, Departamento de Ciencias de la Computación, Aachen, Alemania, 2008.

[159]G. Clarke and J. Wright "Scheduling of vehicles from a central depot to a number of delivery points", Operations Research, 12 #4, 568-581, 1964.

[160] Babin, Gilbert; Deneault, Stéphanie; Laportey, Gilbert (2005), *Improvements to the Or-opt Heuristic for the Symmetric Traveling Salesman Problem*, Cahiers du GERAD, **G-2005-02**, Montreal: Group for Research in Decision Analysis, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.89.9953.

[161] Ioannis Mavroidis, Ioannis Papaefstathiou, Dionisios Pnevmatikatos Hardware Implementation of 2-Opt Local Search Algorithm for the Traveling Salesman Problem Microprocessor and Hardware Lab (MHL)Technical University of Crete (TUC)Kounoupidiana, Crete, GR73100, Greece, 2007.

[162] Perttunen. On the Significance of the inictial solution in traveling salesman heuristics, Journal of the Operational Research Society, 1994.

[163] Zweig, G. Zweig. An Effective Tour Construction and Improvement Procedure for Traveling Salesman Problem. Operations Research, Vol. 43, No. 6, November-December 1995, 1049-1057, 1995.

[164]Leeuwen, J.V., Schoone, A.A. *Untangling a traveling salesman circuito in the plane*, Tech Report , Departamento de Ciencias de la Computación, Universidad de Utrecht, Noviembre de 1980.

[165] Stelios H. Zanakis,James R. Evans,Alkis A. Vazacopoulos. Heuristic methods and applications: A categorized survey Original Research Article European Journal of Operational Research, Volume 43, Issue 1, 6 November 1989, Pages 88-110.

[166] Edmonds, Jack "Paths, trees, and flowers". Canad. J. Math. 17: 449–467. doi:10.4153/CJM-1965-045-4,1965.

[167] MACGREGOR and T. ORMEROD, Human performance on the traveling salesman problem Perception & Psychophysics ,1996, 58 (4), 527–539 J. N. Loughborough University of Technology, Loughborough, England.

[168] Graham SM, Joshi A, Pizlo Z. Memory and Cognition. 2000 Oct;28(7):1191-204. The traveling salesman problem: a hierarchical model. Purdue University, West Lafayette, Indiana, USA.

[169] Iris Van Rooij, Ulrike Stege, and Alissa Schactman. Convex hull and tour crossings in the Euclidean traveling salesperson problem: Implications for human performance studies Memory & Cognition ,2003, 31 (2), 215-220 University of Victoria, Victoria, British Columbia, Canada.

[170] Okano , Shinji Misono , Kazuo Iwano. New TSP Construction Heuristics and Their Relationshipsto the 2-Opt. Journal of Heuristics ,Volume 5 Issue 1, April 1999 ,Kluwer Academic Publishers Hingham, MA, USA.

[171] Du and F. Hwang, editors. Computing in Euclidean Geometry. World Scientific Publishing Co Pte Ltd, 2 edition, 1995.

[172] Rosa Guerequeta y Antonio Vallecillo Técnicas de diseño de algoritmos Servicio de Publicaciones de la Universidad de Málaga. 1998 ISBN:84-7496-666-3 ** Segunda Edición: Mayo 2000 . (Libro electrónico) http://www.lcc.uma.es/~av/Libro/indice.html.

[173] M.Abellanas, D.Lodares. Análisis de Algoritmos y teoría de grafos. Macrobit ra-ma. 1991.

[174]Franco P. Preparata and Michael Ian Shamos (1985). Computational Geometry - An Introduction. Springer-Verlag. 1st edition: ISBN 0-387-96131-3; 2nd printing, corrected and expanded, 1988: ISBN 3-540-96131-3.

[175] Dillard S.E., Natarajan V., Weber G.H., Pascucci V., Hamann B. Topology-guided Tessellation of Quadratic Elements. International Journal of Computational Geometry & Applications (IJCGA), 19(2), pp. 195–211, 2009.

[176] Jon Bentley & Thomas Ottmann, "Algorithms for Reporting and Counting Geometric Intersections", IEEE Trans. Computers C-28, 643-647 (1979)

[177] Bentley, J.L.; Friedman, J.H.: "Data structures for range searching". Computing Surveys, Vol. 11, 4, 397/409, (1979).

[178] [Cormen, 1990] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 1990. ISBN 0-262-03293-7. Section 33.2: Determining whether any pair of segments intersects, pp.934–947.

[179] Michael Shamos & Dan Hoey, "Geometric Intersection Problems", Proc. 17-th Ann. Conf. Found. Comp. Sci., 208-215 (1976)

[180] Wiorkowski y McElvain, 1975. A rapid heuristic for the approximate solution of the traveling salesman problem. Trasportation Research, 9, 1975, pp 181-186.

[181] Or, I., 1976, Traveling salesman type combinatorial problems and their relation to the logistics of blood banking. PhD thesis, Northwestern University,Evanston, Il.1976.

[182] Stewart, W. R., JR, A computationally efficient heuristic for the traveling salesman problem. Proceedings of the 13th Annual Meeting of S. E. TIMS, pp. 75–85, 1977.

[183] John P. Norback, Robert F. Love. Geometric Approaches to Solving the Traveling Salesman Problem. MANAGEMENT SCIENCE, Vol. 23, No. 11, July 1977, pp. 1208-1223.

[184] M. L. Fisher, "Optimal Solution of Vehicle Routing Problems Using Minimum K-trees", Operations Research 42, 626-642, 1994.

[185] J. L. Bentley. Multidimensional binary search trees used for associative searching. Comm. ACM 18(9), 509-517 (Sept. 1975).

[186] A. Lodi, A. Punnen (2002).TSP Software.In G. Gutin, A. Punnen (Eds.).The Traveling Salesman Problems and its Variations,Kluwer Academic Publishers, Dordrecht, 737--749.

[187] D.E. Rosenkrantz, R.E. Stearns and P.M. Lewis. "An Analysis of Several Heuristics for the Traveling Salesman Problem". SIAM Journal on Computing. Vol. 6, pp. 563-581. 1977.

[188] Hansen, P., Mladenovic, N.:First vs best improvement: an empirical study. Discrete Applied Mathematics, 2004; forthcoming.

[189] David S. Johnson and Lyle A. McGeoch. The traveling salesman problem: A case study in local optimization. In E. H. L. Aarts and J. K. Lenstra, editors, Local Search in Combinatorial Optimization. Páginas 215-310, John Wiley and Sons, 1997.

[190] Bajaj, C. Geometric Optimization and Dp Completeness. Discrete and Computational Geometry. Vol. 4, 1989, pp. 3-13.

[191] J.L. Bentley, Experiments on Geometric Traveling Salesman Heuristics. Tech. Report, AT&T Bell Laboratories, Murray Hill, 1990.

[192] Boc, F. An algorithm for solving traveling salesman and related network optimization problems., presented at the fourteen national meeting of the operations research society of america. St. Luis, Missouri, October, 24, 1958.

[193] G. Reinelt. Fast Heuristics for large geometric traveling salesman problems. ORSA J. Comput. Volumen 4: 206-217, 1992.

[194] D.L. Miller y J.F. Pekny. A staged primal-dual algorithm for perfect b-matching with edges capacities. ORSA J. Comput ., Volumen 7, páginas 298-320, 1995.

Rerefencias electrónicas

[WWW1] http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

[WWW2] [http://www.tsp.gatech.edu/pla85900/compute/cpu.htm]

[WWW3] http://www.tsp.gatech.edu/sweden/heur/heur.htm

[WWW4]http://www.slideshare.net/bewatched/presentacin-del-problema-de-asignacin-cuadratica-qap

[WWW5] http://www.cic.ipn.mx/aguzman/papers/89%20Arboles%20kd.htm

[WWW6] http://www.cse.unsw.edu.au/~lambert/java/3d/hull.html