Universidad Nacional Autónoma de México

Facultad de Ciencias

MÉTODOS HEURÍSTICOS EN LA SOLUCIÓN DE PROBLEMAS ENTEROS

T E S I S

QUE PARA OBTENER EL TÍTULO DE:
ACTUARIO

PRESENTA:
ISAAC BUENDÍA HERNÁNDEZ

DIRECTOR DE TESIS:
MA. DEL CARMEN HERNÁNDEZ AYUSO

2016

Ciudad Universitaria, D. F.

1. Datos del alumno
Buendía
Hernández
Isaac
56 70 69 95
Universidad Nacional Autónoma de México
Facultad de Ciencias
Actuaría
307073978

2. Datos de la tutora
M. en I.O.n
María del Carmen
Hernández
Ayuso

3. Datos del sinodal 1
Dra.
Claudia
Orquídea
López
Soto

4. Datos del sinodal 2
M. en I.
Adrián
Girard
Islas

5. Datos del sinodal 3
Act.
Germán
Valle
Trujillo

6. Datos del sinodal 4
M. en B.
Leonardo
López
Monroy

Agradecimientos

Es mi deseo expresar mi agradecimiento hacia mis padres: Maribel y Martín por todo su apoyo y amor, gracias a ustedes he llegado a hasta aquí.

A mi hermana, Saray, por toda la paciencia y apoyo que ha tenido conmigo a lo largo de todos estos años.

Agradezco la amistad y el afecto de mis amigos: Alexis, Francisco, Paulina y especialmente a Nicté por haberme animado a iniciar este proyecto.

También doy gracias por el tiempo de todos mis sinodales: Claudia, Adrián, Leonardo y Germán. Además porque sus comentarios me ayudaron a realizar un mejor trabajo.

Le agradezco a mi asesora, María del Carmen por todas las lecciones que me ha enseñado, por su tiempo, por el apoyo y por su paciencia.

Finalmente a la UNAM, el lugar que para mí representa un segundo hogar: Gracias.

Introducción

Un problema entero es muy similar a uno lineal, la diferencia está en que los primeros tienen como restricción adicional que sus variables sólo toman valores enteros y aunque podría parecer que esta restricción no es muy importante y no afecta mucho es todo lo contrario, no basta con encontrar el óptimo del problema lineal y truncar o redondear para resolver el entero.

El algoritmo simplex es el más popular en la solución de problemas lineales; el funcionamiento de este algoritmo se basa en la revisión de un conjunto específico de soluciones llamadas puntos extremos porque se puede demostrar que de existir el óptimo se encuentra entre ellos, no obstante, en los problemas enteros estos puntos pueden estar fuera de la región factible pues no necesariamente tendrán valores enteros. Sin embargo, una estrategia que podemos utilizar es la de resolver el problema lineal (la versión *relajada* del problema entero), y en caso de que la solución encontrada no sea entera eliminarla de la región factible (junto con otros puntos que de antemano podamos deducir que no serán factibles) y repetir el algoritmo esperando que esta vez sí se encuentre un óptimo con variables enteras, en caso contrario repetir el proceso hasta que suceda[1].
Otra estrategia es revisar soluciones, una por una, de manera inteligente intentando encontrar una cuya función objetivo sea mejor que las demás[2].

El problema de estos algoritmos es que son tardados, el algoritmo de balas puede llegar a enumerar casi todas las soluciones factibles antes de garantizar que ha encontrado el óptimo, y los otros algoritmos básicamente repiten un algoritmo simplex en cada iteración de modo que si el problema relajado se resuelve con uno el entero puede necesitar varios más.

Si el tiempo no es problema podemos utilizar estos algoritmos, pero en caso de que sí lo sea debemos pensar en otras opciones, como en el caso del problema de transporte donde se diseñó un algoritmo especial basado en el simplex; el problema es que esta opción tampoco está disponible en todos los casos, lo que nos obliga a buscar otra alternativa. Si bien nuestro propósito es tomar la mejor decisión posible debemos estar conscientes de que ésta no siempre es el óptimo,

[1] ejemplos de algoritmos que utilizan esta estrategia son el de ramificación y acotamiento y el de planos de corte

[2] Como es el caso del algoritmo de balas

podemos permitirnos sacrificar la precisión y obtener una solución cercana con tal de tenerla en un tiempo razonable, que es justo lo que hacen los métodos heurísticos.

Los métodos heurísticos están basados en estrategias lógicas, fáciles de implementar y que pueden convencernos de que la solución encontrada debería ser buena, de esta manera su funcionamiento deberá ser más rápido que el de un algoritmo como los que describimos antes, por otro lado, no debemos olvidar que tienen fallas y no garantizan que se encuentre el óptimo. Para dejar una mejor idea de estos métodos y su funcionamiento analicemos el siguiente caso:

Pensemos que se arrojará una moneda cargada que tiene una probabilidad de 0.9 de caer en águila y se nos pide que adivinemos el resultado. Al no tener posibilidad de saber el resultado lo más lógico es inclinarnos por el caso más probable y elegir águila, no obstante es posible que fallemos y el resultado sea sol; más aún, si el ejercicio lo repitiéramos una cantidad mayor de veces la mejor alternativa en cada caso es elegir águila, ya que aún sabiendo de que en algún momento debe caer sol no sabemos cuándo y eligiendo águila garantizamos acertar la mayoría de las veces. Básicamente así funcionan los heurísticos, casi siempre tendrán un resultado bueno y pocas veces será lo contrario, o al menos ese es el propósito.

El objetivo de este trabajo es tratar el tema de los métodos heurísticos, aspectos generales sobre las propiedades que deben tener para que en efecto resulten ser buenos, dar una clasificación, así como la descripción de la estrategia que ocupa cada método y su aplicación en algunos de los problemas enteros revisando dónde fallan y en qué casos se tiene un desempeño aceptable.

Pero antes de llegar a eso, en el Capítulo 1 hablaremos sobre otro punto importante: la complejidad de los problemas. Mencionamos que estos problemas son tardados de resolver, pero debemos dar una justificación más rigurosa del porque no se puede diseñar un algoritmo que sea más eficiente, por lo que daremos los conceptos necesarios para medir el tiempo de ejecución de un algoritmo, las características de un problema difícil y finalmente la demostración de que algunos problemas enteros poseen estas características.

Índice general

Capítulo 1

Teoría de Complejidad e Introducción a los Métodos Heurísticos

Los métodos heurísticos son una alternativa para resolver problemas difíciles debido a que éstos no se pueden resolver en un tiempo razonable con los algoritmos clásicos por lo que es necesario caracterizar este tipo de problemas. Asimismo, esta dificultad se relaciona con el concepto de algoritmo "eficiente" de manera que será necesario comentar la forma en que se mide esta eficiencia.

Para ello tendremos que usar conceptos como el de instancia de un problema y el de algoritmo; la primera porque cada uno de éstos tiene varias instancias (o ejemplos) y son las que el algoritmo resuelve; la segunda porque si queremos medir su eficiencia debemos saber cómo funcionan. Posteriormente describiremos a los problemas de decisión y la relación que tienen con los de optimización.

1.1. Algoritmos y Complejidad

Un algoritmo es un conjunto de instrucciones ordenadas y bien definidas que debemos interpretar para resolver un problema y de éstas dependerá el tiempo que tardemos en hacer esto; por lo tanto empezaremos con la definición de algoritmo, que Sakarovith menciona en [7]:

Definición 1.1 *Un algoritmo para resolver un problema P es un procedimiento que se puede descomponer en operaciones elementales, transformando una cadena de caracteres que representan los datos de P en una cadena de caracteres que represente la solución del problema.*

Un problema de optimización se compone de un conjunto de instancias (o ejemplos) y a su vez cada instancia se compone de dos elementos (F_P, z) donde

F_P es la región factible y z la función objetivo; un algoritmo que esté diseñado para resolver un problema debe resolver todas las instancias del mismo sin importar su tamaño. Estas instancias son las que aparecen en la definición de algoritmo como "problema P". Por ejemplo, un tipo de problema es el que aparece a continuación:

$$\begin{array}{l} Max\ z = c_1x_1 + c_2x_2 \\ s.a \\ a_1x_1 + a_2x_2 \leq b \\ x_i \geq 0 \end{array}$$

Siendo una instancia (P) la siguiente:

$$P = \begin{cases} Max\ z = 2x_1 + 3x_2 \\ s.a \\ 2x_1 + 4x_2 \leq 5 \\ x_i \geq 0 \end{cases}$$

Cuando hablamos del "tamaño" de la instancia nos referimos a la longitud de la cadena inicial; esta cadena está formada por elementos que pertenecen a un alfabeto fijo y finito que utilizamos para codificar nuestro problema (un ejemplo es pasar los datos a bits para que puedan ser interpretados en una computadora).

Sin embargo este concepto de tamaño no es el más utilizado, se puede utilizar (y de hecho nosotros utilizaremos) un número más representativo del problema como el número de variables que tiene (en la instancia que dimos podemos definir su tamaño como 2) o el número de nodos o el de aristas (o arcos) si es un problema que se represente por medio de una gráfica. La importancia del tamaño está en que daremos el tiempo de ejecución del algoritmo en función de éste pues tiene sentido que a mayor número de variables, por citar un ejemplo, el tiempo de resolución sea mayor.

Lo siguiente que necesitaremos es hablar sobre la unidad de tiempo a utilizar; si resolvemos un problema a mano es casi seguro que será más tardado que hacerlo en una computadora pero aún usando computadoras se obtendrá un resultado diferente dependiendo de la capacidad que tenga cada una aún usando el mismo algoritmo, esto implica que el tiempo va a cambiar según el objeto donde implementemos el algoritmo.

Según un principio de invarianza este tiempo diferirá en una constante (ya que el número de operaciones elementales que realiza una máquina en una unidad de tiempo diferirá en una constante respecto al de otra) lo que quiere decir que si un algoritmo funciona en 10 segundos y la constante es 4, el otro se terminará en a lo más 40.

La importancia de este principio está en que no es necesario tener una unidad de medida específica; si tenemos un algoritmo que resuelve un problema de tamaño n en un tiempo dado $kf(n)$ en una computadora y cambiamos por otra mejor sabemos que lo único que cambiará será el valor de la constante k ($f(n)$ es una función que depende del tamaño del problema n); en otras palabras la unidad de medida no es tan relevante pues lo que nos interesa es la curva que tiene la función y que no se ve afectada por una constante por lo que contaremos el número de operaciones elementales que se realizan por unidad de tiempo.

Por otra parte, dos instancias de un problema del mismo tamaño no necesariamente requerirán el mismo tiempo para resolverse; por ejemplo si tenemos un caso de maximización cuyos coeficientes de costo reducido sean menores o iguales a cero no requerirá el mismo tiempo que otro con el mismo número de entradas pero con al menos un coeficiente mayor a cero, de manera que realizar una medición exacta puede no ser la mejor opción y sabiendo que un algoritmo debe resolver todas las instancias de un problema lo mejor es buscar una cota superior al tiempo de ejecución de todas las instancias de un problema de un mismo tamaño (digamos n) es decir, tomaremos el peor de los casos como cota.

Finalmente, debemos hablar sobre las operaciones elementales que son las que el algoritmo realizará en una unidad de tiempo, en otras palabras, una operación elemental es aquella cuyo tiempo de realización sólo depende de la máquina utilizada lo que implica que podemos acotarla superiormente por una constante; algunos ejemplos de este tipo de operaciones son las aritméticas como sumas o restas, así como instrucciones de comparación como mín(a,b), etc.

Con estos elementos podemos emprender con el análisis de algoritmos empezando por caracterizar a los algoritmos eficientes.

1.2. Algoritmos polinomiales

Una vez que conocemos los conceptos necesarios para poder acotar el tiempo de ejecución de un algoritmo procederemos a caracterizar a los que son capaces de resolver un problema en un tiempo razonable, empezando por definir el significado de esta expresión.

Este concepto apareció en 1965 cuando J. Edmonds utilizó el término de algoritmo "bueno" para llamar a aquellos que tienen por cota superior para el número de operaciones elementales que realiza, una función polinomial que depende del tamaño del problema. Las siguientes definiciones expresan de manera formal esta idea:

Definición 1.2 *Sean f,g dos funciones tales que f,g:* $\mathbb{N} \mapsto \mathbb{N}$. *Se denota como O(g) a una función f si existe una constante c tal que:*

$$|f(n)| \leq c|g(n)| \quad \forall\ n \in \mathbb{N}$$

Una función f se dice *polinomial* si es $O(g)$ de g, con g un polinomio. O si existen dos constantes c y k tales que:

$$f(n) \leq cn^k \quad \forall\ n \in \mathbb{N}$$

Definición 1.3 *Un algoritmo se dice polinomial si el número de operaciones elementales necesarias para resolver un ejemplo de tamaño n es una función polinomial de n.*

Se dice que un algoritmo es *eficiente* si y sólo si es polinomial. A diferencia de otras funciones los polinomios crecen más "lento", esto se refleja en la diferencia que hay entre el número de operaciones elementales que se necesitan para resolver un problema de tamaño n y otro de $n+1$, así como la mejora que se tiene si se utiliza una máquina mejor.

Podemos comparar esta diferencia con los siguientes ejemplos:

Supongamos que tenemos el problema de ordenar n números $\{p_1, p_2, ..., p_n\}$ de manera creciente para lo cual usaremos un algoritmo conocido como "*algoritmo de burbuja*" que consiste en comparar cada pareja de elementos adyacentes (p_{i-1} y p_i o p_i con p_{i+1} por ejemplo), si esta pareja está ordenada ($p_i < p_{i+1}$) pasamos a la siguiente, sin embargo, si esto no sucede hacemos un intercambio. A continuación mostramos un ejemplo de cómo hace la comparación el algoritmo.

$\{\mathbf{1}, \mathbf{3}, 2\}$ Comparamos los primeros 2 números

$\{1, \mathbf{3}, \mathbf{2}\}$ Comparamos los números 2 y 3

$\{1, \mathbf{2}, \mathbf{3}\}$ Hacemos un intercambio para que los números 2 y 3 queden en el orden que deberían

Podemos observar que en este caso los números fueron ordenados cuando comparamos todas las parejas, pero esto no siempre pasa a la primera, por ejemplo, si hubiéramos teniendo la secuencia $\{3, 2, 1\}$ terminaríamos con $\{2, 1, 3\}$. Lo que siempre ocurre es que el último elemento sí queda en el lugar correcto, por lo que si realizamos otra comparación sólo incluiremos a los primeros 2 elementos $\{2, 1\}$ con lo que 2 ya quedará en su lugar y tendremos a los números ordenados $\{1, 2, 3\}$.

Por lo tanto, en el peor de los casos el algoritmo necesitará $n-1$ iteraciones; por otro lado, en cada iteración disminuyen el número de comparaciones que se hacen siendo $n-1$ en la primera, $n-2$ en la segunda y en general $n-i$ en la i-ésima, esto es:

$$(n-1) + (n-2) + ... + 1 = \frac{n(n+1)}{2}$$

También serán necesarios como máximo el mismo número de intercambios (suponiendo que en cada comparación fuera necesario hacer el intercambio) con lo que llegamos a:

$$n(n-1) = n^2 - n$$

Esto implica que el algoritmo es polinomial pues es $O(n^2)$.

Otra manera de ordenar un conjunto de números es con el algoritmo Radix que utiliza como estrategia el agrupar el conjunto según su última cifra (es decir, el dígito que indica las unidades) y ordenando a los elementos con esa cifra en común, por ejemplo, los números 453,123,255 se agruparían primero en (12**3**, 45**3**) y (255). Una vez que han sido ordenados según esa cifra (en nuestro caso sería 123,453 y 255) los agrupamos por su penúltima cifra (la que indica las decenas), teniendo (1**2**3) y (**2**5**5**, 4**5**3) y ordenamos: 123,255,453. Este algoritmo se repite tantas veces como el número de dígitos que tengan los números que estamos ordenando, en este caso serán 3 pues cada número tiene 3 dígitos.

Cabe mencionar que si los números tienen una cantidad de dígitos diferente basta con añadir tantos ceros como el número de dígitos que tenga el número más grande, por ejemplo, si tenemos que ordenar los siguientes números: 12,698,123,4 debemos empezar con 012,698,123 y 004.

A modo de ejemplo aplicaremos el algoritmo con estos números. Orden inicial: 122,618,012,004,008.
Orden según el *dígito de unidades*:
(01**2**,12**2**),(00**4**),(00**8**,61**8**)
Nuevo orden:
012,122,004,008,618
Orden según el *dígito de decenas*:
(0**0**4,0**0**8),(0**1**2,6**1**8),(1**2**2)
Nuevo orden:
004,008,012,618,122
Orden según el *dígito centenas*:
(**0**04,**0**08,**0**12),(**1**22),(**6**18)
Orden final:
004,008,012,122,618

Podemos observar que en cada iteración se hacen n movimientos para formar los grupos y para ordenarlos según su grupo son necesarios otras n comparaciones dando un total de $2n$ y éstas se repiten tanto como en el número de dígitos d que tengan nuestros números. Por lo tanto necesitamos $2(nd)$ movimientos, esto implica que este algoritmo es $O(n)$.

No obstante no todos los algoritmos son polinomiales como lo muestra el siguiente ejemplo: Supongamos que ahora debemos resolver el siguiente problema:

$$Max\ z = c_1x_1 + c_2x_2 + ... + c_nx_n$$
$$s.a$$
$$a_1x_1 + a_2x_2 + ... + a_nx_n \leq b$$
$$x_i \in \{0, 1\}$$

Para lo cual usaremos un algoritmo basado en las siguientes instrucciones:

1. Sea $F = \{s \mid s = \{x_1, x_2, ..., x_n\}$ con $x_i \in \{0, 1\}\}$
2. Para cada $s \in F$ calcular $p = a_1x_1 + a_2x_2 + ... + a_nx_n$
3. Verificar si $p \leq b$ y de ser así calcular $z(s) = c_1x_1 + c_2x_2 + ... + c_nx_n$, en caso contrario regresar a 2.

Para los pasos 2 y 3 el número de operaciones elementales es $O(n)$ ya que en cada paso se realizan n sumas y n multiplicaciones, que dan un total de $4n$ (En el paso 2 hay n multiplicaciones del tipo a_nx_n y hay un número similar de sumas del estilo $a_ix_i + a_jx_j$ lo que da un total de $2n$ operaciones; para calcular el valor de la función objetivo es un número similar),. El problema es el paso 1 ya que si son muchas las soluciones que se construyen el algoritmo puede dejar de ser polinomial. Tenemos n variables, cada una de las cuales puede tomar dos valores 0 o 1, esto implica que tenemos 2^n posibles soluciones, por ejemplo para $n = 2$ tenemos $2^2 = 4$ soluciones $(0, 0), (0, 1), (1, 0), (1, 1)$ que dan un total de $4n2^n$ operaciones elementales, por lo tanto el algoritmo no es polinomial.

1.3. Problemas de decisión

Siguiendo con los elementos necesarios para caracterizar a los problemas difíciles pasaremos al tema de los problemas de decisión que se definen de la siguiente manera:

Definición 1.4 *Un problema de decisión es aquel en el que el resultado es uno de los dos valores: Verdadero o Falso.*

Quiere decir que son enunciados cuya veracidad, o falsedad, debemos probar. Por ejemplo la pregunta ¿el número p no es primo? cuya respuesta puede ser *sí* (o verdadero) cuando p no es un número primo, o *no* en caso contrario.

El siguiente problema también es de decisión y será de nuestro interés porque fue el primer problema que se demostró como difícil; éste es el SAT o problema de *satisfacibilidad*[1] cuyo enunciado está compuesto por una serie de cláusulas:

$$E = C_1 \wedge C_2 \wedge ... \wedge C_m$$

Las cuales deben ser ciertas (todas) para que el enunciado E lo sea; a su vez C_i representa una expresión del siguiente tipo:

$$C_i = u_{i_1} \vee u_{i_2} \vee ... \vee u_{i_m}$$

[1] En inglés satisfacibilite

Lo que implica que cada cláusula es verdadera si al menos un u_{ij} es verdadero; cada una de éstas es representada por una variable $x_i \in 0, 1$
$foralli \in X$, que es un elemento de un conjunto $X = \{x_1, x_2, ..., x_n\}$, o su complemento $(\overline{x}_i 01 - x_i)$.

En resumen el problema SAT es un enunciado compuesto por variables de X o el complemento de éstas, agrupadas en cláusulas que deben ser todas verdaderas para que éste sea verdadero. Esto implica que la respuesta al enunciado se obtiene a partir de una combinación de las variables x_i (conociendo éstos valores los complementos también quedan determinados).

A modo de ejemplo presentaremos el siguiente problema SAT que tiene 3 cláusulas:

$$E = (x_1 \vee \overline{x}_2 \vee x_3) \wedge (\overline{x}_1 \vee x_3) \wedge (x_2)$$

El cual tiene 3 variables distintas por lo que $X = \{x_1, x_2, x_3\}$, para saber si es verdadero debemos encontrar al menos una combinación que haga ciertas todas las cláusulas como la siguiente: $x_1 = x_2 = x_3 = 1$ no necesariamente única pues aquí la respuesta $x_1 = 0$ y $x_2 = x_3 = 1$ también hace verdadero al enunciado.

En cambio, para que sea falso no debe existir alguna combinación de variables que lo haga verdadero como el enunciado resultante al añadir la cláusula $C_4 = (\overline{x}_2 \vee \overline{x}_3)$:

$$E = (x_1 \vee \overline{x}_2 \vee x_3) \wedge (\overline{x}_1 \vee x_3) \wedge (x_2) \wedge (\overline{x}_2 \vee \overline{x}_3)$$

Como ya adelantamos la caracterización de problemas de optimización difíciles está relacionada con los de decisión, esto es porque podemos asociar un problema de decisión con uno optimización, tal como podemos ver con un Problema de Programación Lineal Entero de la forma:

$$\begin{aligned} &Min\ z = cx \\ &\text{s.a} \\ &Ax = b \\ &x_i \geq 0,\ x_i \text{ entero} \end{aligned}$$

Y un entero positivo k con lo que podemos plantear el siguiente enunciado: ¿Existe una solución factible tal que $z(x) \leq k$?, enunciado que es un problema de decisión.

Pero esta relación no sería suficiente si no se extendiera también a la dificultad que tienen para resolverse; esto es justo lo que el siguiente teorema nos proporciona:

Teorema 1.1 *Si el problema de decisión asociado a un problema de optimización combinatoria dado es difícil, entonces el problema de optimización combinatoria es igualmente difícil, es decir, el problema de decisión es al menos tan fácil como el problema de optimización asociado.*

Demostración. Para resolver el problema de decisión es necesario resolver el problema de optimización para después comparar la solución obtenida en este problema con la condición que se establece en la pregunta del problema, verificar esta condición puede dificultar la resolución del problema (en ningún caso la hará más fácil), por lo que el problema de decisión debe ser tan fácil como el problema de optimización.

1.4. Las clases P, NP y NP-C

Finalmente, llegamos a la caracterización de los problemas con el objeto de reconocer a los difíciles; como mencionamos en la propiedad **??** los algoritmos polinomiales resuelven de manera eficiente a los problemas, por lo que aquellos problemas de decisión que pueden ser resueltos por un algoritmo polinomial son fáciles, diremos que éstos pertenecen a la clase P.

Definición 1.5 *Se dice que un problema de decisión pertenece a la clase P si el problema puede ser resuelto por un algoritmo polinomial.*

La siguiente clase que definiremos será la clase NP, que es la clase que contiene a los problemas fáciles y a los difíciles (entre otros). Una idea no informal de la clase NP la tenemos apoyándonos en el concepto de "supervisor". Pensemos en un problema de decisión cuya respuesta es Verdadero, si podemos convencer a otra persona (al supervisor) de la validez de dicha respuesta en tiempo polinomial, entonces pertenece a la clase NP. Un hecho importante a notar es que el principio del supervisor se aplica aún si no sabemos si es posible encontrar en tiempo polinomial una solución s para la cual la respuesta del problema sea verdadera. El principio sólo pide que si la solución s es propuesta, sea posible verificar en tiempo polinomial que la respuesta correspondiente sea verdadera.

El problema SAT es un ejemplo de problema NP, si tenemos una combinación de valores de las variables x_i para las cuales el valor de E es igual a 1, basta con calcular el valor correspondiente de E para convencer al supervisor. En el caso del problema del agente viajero[2] , para convencer al supervisor de la existencia de un ciclo hamiltoniano basta con mostrar ese ciclo.

Los problemas NP toman su nombre del hecho de que pueden ser resueltos por un algoritmo no determinista en tiempo polinomial (Nondeterministic Polinomial Time en inglés). Un algoritmo no determinista es una construcción abstracta que a diferencia de uno determinista no puede ser programado en una

[2]El problema del agente viajero consiste en encontrar, en una gráfica G, un ciclo hamiltoniano que recorra la gráfica con la menor distancia posible.

computadora. Esto es porque los algoritmos no deterministas hacen uso de una operación cuyo criterio de elección no es preciso, mediante esta operación se elige un elemento de un conjunto finito pero no especifica de manera precisa cuál es el criterio utilizado para elegir dicho elemento.

En los problemas de decisión, dependiendo de la forma en que se efectúe la operación de elección, la respuesta podría ser Verdadero o Falso. En el caso de los algoritmos no deterministas, hay al menos una manera de tomar la decisión (si existe) que dé como resultado una respuesta verdadera, el algoritmo tomará la decisión mediante dicha manera.

La definición que consideraremos de la clase NP es la siguiente:

Definición 1.6 *Un problema pertenece a la clase NP si es posible resolverlo, en tiempo polinomial, usando un algoritmo no determinista.*

Por ejemplo:

Supongamos que nuestro problema consiste en responder si es posible ordenar n números: $a_1, a_2, ..., a_n$ de manera creciente. Sabemos que siempre será posible hacer este ordenamiento, pero supongamos que la única forma de responder el problema es construyendo una permutación $p = \{p_1, p_2, ..., p_n\}$ de {1,2,3,...,n} tal que:

$$a_{p_1} \leq a_{p_2} \leq ... \leq a_{p_n}$$

Entonces, un algoritmo para construirla:

Sea $S = \varnothing$

Para $k = 1$

$p_1 =$ "*elegir*"$(\{1, 2, 3, ..., n\})$ y sea $S = \{p_1\}$. Hacer $k = k + 1$

Para $k = 2, 3, ..., n$:

$p_k =$ "*elegir*"$(\{1, 2, 3, ..., n\} \backslash S).S = S \bigcup p_k$

Si $a_{p_k - 1} > a_{p_k}$ entonces la respuesta es Falso.

En caso contrario hacer $k = k + 1$. Si $k \leq n$ repetir la instrucción.

En caso contrario la respuesta es Verdadero.

Fin del algoritmo.

En la práctica, la implementación de un algoritmo no determinista es complicada debido a que la instrucción de elección que poseen no siempre es muy clara como para implementarla computacionalmente, la idea de esta instrucción es que tome la mejor decisión basándose en las decisiones anteriores, sin embargo, en la teoría son muy útiles ya que nos sirven para definir esta clase de problemas.

Otra clase de problemas es aquella cuyos elementos pueden ser resueltos de manera eficiente si suponemos que tenemos un algoritmo eficiente para resolver un segundo problema que nos ayude a resolver el primero. Éstos son los pertenecientes a la clase NP-C. Para caracterizar a los problemas NP-C primero daremos el concepto de reducción polinomial.

Definición 1.7 *Sean P_1 y P_2 dos problemas de decisión. Decimos que P_1 se reduce en tiempo polinomial a P_2 si y sólo si existe un algoritmo $\mathfrak{a}_1$ para P_1 que tome como una sub rutina (que se cuenta como una operación elemental) un algoritmo $\mathfrak{a}_2$ para P_2. Llamamos a $\mathfrak{a}_1$ una reducción polinomial de P_2 a P_1.*

Un punto importante de esta definición es la hipótesis de considerar la resolución de P_2 con $\mathfrak{a}_2$ como una operación elemental, pues esta no es una suposición muy realista. El siguiente teorema obtiene un resultado importante cuando $\mathfrak{a}_2$ es polinomial.

Teorema 1.2 *Si P_1 se reduce polinomialmente a P_2 y existe un algoritmo polinomial para resolver P_2, entonces existe un algoritmo polinomial para resolver P_1*

Demostración. Supongamos que $p_1(n)$ es el polinomio que acota el tiempo de resolución de $\mathfrak{a}_1$ (si consideramos a $\mathfrak{a}_2$ como una operación elemental) y que $p_2(n)$ es el polinomio que acota a $\mathfrak{a}_2$. Entonces el tiempo real de resolución de $\mathfrak{a}_1$ (si se tiene una entrada de n elementos) está acotado por:

$$p(n) = p_1(n) * p_2(p_1(n))$$

Para ver esto, notemos que en el peor de los casos $\mathfrak{a}_1$ consistirá en llamar repetidas veces a $\mathfrak{a}_2$, lo cual sucederá a los más $p_1(n)$ veces, ahora, falta ver qué tan largo pudiera llegar a ser el número de entradas de $\mathfrak{a}_2$. De nuevo, si consideramos el peor de los casos $\mathfrak{a}_1$ podría ocupar sus $p_1(n)$ pasos en generar las entradas para $\mathfrak{a}_2$, es decir, que el número de entradas para $\mathfrak{a}_2$ sería a lo más de $p_1(n)$. Por lo tanto, el tiempo de resolución de P_1 es polinomial.

La relación de reducción polinomial es transitiva. Esta propiedad nos permitirá demostrar la reducción polinomial de un problema a otro sin necesidad de hacerlo directamente pues podremos usar un problema intermedio.

Teorema 1.3 *Si P_1 se reduce polinomialmente a P_2 y P_2 se reduce polinomialmente a P_3. Entonces P_1 se puede reducir polinomialmente a P_3*

Finalmente podemos dar una definición de problemas difíciles.

Definición 1.8 *Un problema de decisión se dice NP completo si todo problema de la clase NP puede ser reducido polinomialmente a él.*

Por el teorema 1.2 los problemas NP-C tienen una propiedad muy particular: Si existe un algoritmo eficiente para resolver un problema NP-C, entonces existe uno para todo problema de la clase NP-C.

Para demostrar que un problema está en la clase NP-C tenemos que probar lo siguiente:

1 El problema pertenece a la clase NP

2 Todos los problemas pertenecientes a la clase NP se pueden reducir polinomialmente a este.

La demostración del punto 2 de manera directa resulta un tanto complicada ya que para empezar ni siquiera conocemos a todos los problemas de la clase NP, sin embargo, se puede resolver más fácilmente utilizando el resultado obtenido con el teorema 1.3 de esta manera el punto 2 se reduce a demostrar que un problema NP-C conocido se puede reducir polinomialmente a nuestro problema.

Sin embargo, demostrar que la clase NP-C es distinta del vacío se necesitó hacer la demostración directa del punto 2, esta demostración fue presentada por S. A. Cook quién demostró que el problema SAT pertenece a la clase NP-C.

Aunque no incluiremos la demostración realizada por Cook, hemos agregado un anexo donde se pueden consultar las demostraciones de que los siguientes problemas pertenecen a la clase NP-C:

- **El problema 3-SAT**. Que es un caso particular del problema SAT con la condición adicional de que cada cláusula debe tener exactamente 3 variables (es decir, cada $C_i = \{u_{i1} \vee u_{i2} \vee u_{i3}\}$).

- **El problema de la cubierta exacta**. Dada una familia $S = \{S_i\}$ ¿Existe una subfamilia $W \subseteq S$ tal que los $W_i \in W$ sean ajenos y $\cup W_i = \cup S_i$.

- **El problema del clique de cardinalidad máxima**. Dada una gráfica $G = (X, A)$ donde X es el conjunto de nodos y A el de aristas un *clique* C es un conjunto de nodos tales que cada nodo de C es adyacente a los demás nodos de C. El problema de optimizaación consiste en encontrar el clique con la mayor cardinalidad.

- **El problema del número cromático**. Este problema consiste en colorear los nodos de una gráfica con la menor cantidad de colores de manera que cada nodo tenga un color diferente al de sus vecinos.

- **El problema binario de la mochila**. En este problema se busca maximizar una función objetivo sin que se sobrepase de cierta capacidad, es decir, se debe cumplir la restricción $\sum a_j x_j \leq b$.

- **El problema de secuencia de tareas**. Dado un conjunto de trabajos $\{1, 2, ..., t\}$ con tiempos de realización $\{\tau_1, \tau_2, ..., \tau_t\}$ enteros, un conjunto de fechas (también entero) $\{d_1, d_2, ..., d_t\}$ y un vector de penalización $\{P_1, P_2, ..., P_t\}$ buscamos un ordenamiento de los trabajos $\pi = (\pi_1, \pi_2, ..., \pi_t)$ tal que el tiempo de penalización sea mínimo.

Además incluimos una lista con otros problemas que pertenecen a la clase NP-C. Elegimos estas demostraciones porque son algunos de los problemas que utilizaremos para aplicar los métodos heurísticos que describiremos posteriormente.

1.5. Problemas NP-Duros

Si un problema no pertenece a la clase NP, pero es posible hacer una reducción polinomial de todos los elementos pertenecientes a la clase NP, no es posible clasificarlo como NP Completo, sin embargo, la complejidad que se tiene para resolverlo es equivalente a la de éstos. A este tipo de problemas lo llamaremos **NP-Duros**[3]. Es de esperarse que los problemas de optimización asociados a los problemas de decisión NP-Completos pertenecen a esta clase, pues ya hemos demostrado que son igual de difíciles de resolver que los de decisión asociados y al no ser problemas de decisión no pueden ser incluidos en la clase NP. Por lo que a partir de ahora nos referiremos a los problemas difíciles de resolver como NP Completos y NP Duros según sea el caso.

Los problemas NP Duros no pueden ser resueltos en un tiempo polinomial por un algoritmo. Ante esta situación se tienen 3 posibilidades para resolver el problema:

- Si el problema no se puede resolver en tiempo polinomial pero es posible resolverlo en un tiempo menor al que tenemos disponible podemos usar el algoritmo que tenemos.
- Podemos aprovechar alguna estructura especial que tenga el problema para diseñar un algoritmo especial que permita reducir el tiempo en su resolución.
- Si no es posible ninguno de los puntos anteriores la opción es buscar una solución "buena" en un tiempo razonable, es decir, una aproximación al óptimo.

En el tercer caso se recurre a un algoritmo que no resuelve el problema en el sentido de que encuentre el óptimo, pero es capaz de darnos una solución que puede estar cerca del óptimo, incluso en ocasiones puede llegar a coincidir. Estos algoritmos son los métodos heurísticos.

1.6. Introducción a los Métodos Heurísticos

Una vez caracterizados los problemas difíciles es tiempo de buscar soluciones, ya que saber que un problema es difícil no facilita su solución. De las alternativas que describimos al término de la sección anterior nos concentraremos en la última debido a que para problemas difíciles la construcción de un algoritmo especializado tampoco podrá resolverse en tiempo polinomial.

Aquí es donde entran los métodos que trataremos en los próximos capítulos: los heurísticos; que se caracterizan por encontrar una solución cercana al óptimo en un tiempo razonable basándose en estrategias lógicas mediante las que

[3]NP-Hard en inglés

se excluyen soluciones malas y se examinan soluciones prometedoras.

Estas estrategias no realizan un análisis completo de la región factible, lo que ocasiona que no exista la certeza de que el óptimo sea encontrado; no obstante, su desempeño es bueno y existen varios problemas fáciles que puede demostrarse que sí lo hacen como con el problema de árbol de peso mínimo.

La aplicación de los heurísticos tiene más ventajas que la posibilidad de encontrar una solución para problemas de programación lineal entera; por ejemplo, para resolver problemas no lineales (enteros) ya que los heurísticos pueden utilizarse para problemas de optimización combinatoria[4]. Incluso pueden aplicarse en la resolución de problemas fáciles que se necesiten resolver en un tiempo menor, donde sólo nos interese saber si existe alguna solución que rebase alguna cota.

A diferencia de los algoritmos, los heurísticos son instrucciones más generales lo que permite hacerlos más personalizables y como consecuencia, el mismo método deberá ser distinto para dos tipos de problemas difíciles. Esto quiere decir además que el desempeño de cada heurístico será diferente para cada problema; habrá casos donde se encuentren soluciones tan cercanas al óptimo que incluso en varios puede llegar a coincidir y que al ser aplicados a otro problema se obtenga la peor solución de todas.

Aún así, de manera general la estrategia empleada por un método es buena casi siempre o de eso debemos asegurarnos antes de implementarlo en un problema. Para poder considerar a un método como adecuado debe cumplir 3 características:

- **Bondad.** La solución que presente el método en la mayoría de los casos debe ser cercana al óptimo
- **Eficiencia.** La estrategia diseñada debe proporcionar una solución a tiempo, de lo contrario el método pierde su objetivo
- **Robustez.** Hemos mencionado que el método puede fallar, sin embargo, tales fallas deben ser pocas y poco graves, esto es que la probabilidad de recibir una mala solución por parte del heurístico debe ser baja.

Una desventaja de estos métodos es que aunque de manera intuitiva se observa que las soluciones son buenas y en consecuencia "cercanas" al óptimo, pero en realidad desconocemos qué tan cercanas son. Si tuviéramos conocimiento sobre el valor de la solución óptima, una posibilidad para medir la calidad del heurístico consistiría en calcular la diferencia entre las soluciones como con la siguiente fórmula:

[4]Es decir, aquellos problemas donde el conjunto de soluciones S es un conjunto finito, aunque generalmente no son pequeños y donde la función objetivo y las restricciones no necesariamente deben ser lineales

$$\frac{|f(\tilde{x}) - f(x^*)|}{f(x^*)}$$

Donde $\tilde{x}$ sería la solución obtenida por el método y x^* la óptima.

Desafortunadamente no conocemos a x^* de manera que debemos emplear otra estrategia. Por ejemplo, comparar con los resultados obtenidos con otro método.

Otra razón por la que los métodos heurísticos no encuentran la solución óptima es porque la estrategia en la que están basados es propensa a fallar en situaciones específicas. Motivo por el cual se han desarrollado más métodos, varios de ellos variantes inspirados en otros conocidos como *metaheurísticos*, que por medio de instrucciones adicionales disminuyen la probabilidad de fallar en los casos donde los otros sí; instrucciones sobre como actuar en estas situaciones o continuar con algunas iteraciones para verificar que realmente se haya encontrado una buena solución. Otra alternativa aplicada es la de utilizar la combinación de varios métodos para aumentar la calidad de una solución encontrada. Todos éstos métodos siguen siendo considerados como heurísticos.

Cabe mencionar que con estos cambios tampoco está garantizado el hallazgo del óptimo. Aunque se pretende eliminar las fallas y mejorar la calidad de la solución puede que se termine encontrando la misma solución que con un heurístico más básico lo que se resume en un aumento de recursos sin mejorar la calidad.

Los métodos que trataremos se pueden clasificar en dos grupos:

- **Métodos constructivos.** Son aquellos que aplican como estrategia para elaborar soluciones el agregar candidatos paso a paso.
- **Métodos de búsqueda local.** Son los métodos que analizan una vecindad en tratando de encontrar mejores soluciones, a partir de una solución inicial y mediante un mecanismo de intercambio es como estos métodos avanzan en cada iteración.

A diferencia de los métodos de búsqueda local los constructivos no exploran soluciones vecinas, de manera que la estrategia con la que eligen el siguiente candidato a conformar la solución es la parte de mayor importancia y la que determinará la calidad de la solución. Los heurísticos constructivos que describiremos son el *Glotón* y el de *Colonia de Hormigas*. El primero es el más básico y fácil de implementar pues sólo construye una única solución esperando que sea buena añadiendo en cada iteración al mejor candidato que pueda. Colonia de Hormigas funciona diferente pues imita una estrategia empleada por las hormigas en la búsqueda y traslado de alimento de modo que construye varias soluciones "marcando" a los candidatos según la calidad de la solución para que las iteraciones posteriores encuentren soluciones de mejor calidad.

Los métodos de búsqueda local que revisaremos son 3: *Mejoras Sucesivas, Recocido Simulado y Búsqueda Tabú*. De éstos, los últimos dos son variantes del primero, es decir, son metaheurísticos diseñados para prevenir la principal falla de los métodos de búsqueda local que es la caída en óptimos locales. Al moverse dentro del espacio de soluciones mediante un mecanismo de intercambio los métodos de búsqueda local dependen de un mecanismo de intercambio mediante el cual puedan acceder de una solución a otra. Teniendo como desventaja el revisar soluciones más de una vez si éstas son vecinas de dos soluciones distintas, por lo que se suelen usar listas o alguna manera más eficiente de hacer estas comprobaciones.

Capítulo 2

Métodos Constructivos

Los métodos heurísticos basan su estrategia en la elección de candidatos para formar la solución, es decir construyen soluciones paso a paso añadiendo candidatos factibles hasta completar una solución.

Supongamos que una solución puede escribirse como un conjunto $C = \{c_1, c_2, ..., c_n\}$ donde cada c_j es un candidato; los métodos constructivos eligen a cada c_j del conjunto $\tilde{C}$ que contiene a todos los candidatos posibles.

En los problemas representados por una gráfica es más fácil hacer este proceso ya que en la mayoría de éstos buscamos generalmente un conjunto de arcos, nodos o aristas. Por ejemplo, si en la figura 2.1 queremos determinar una ruta de I a F, la solución es representada como un conjunto de aristas que debemos cruzar para llegar a F.

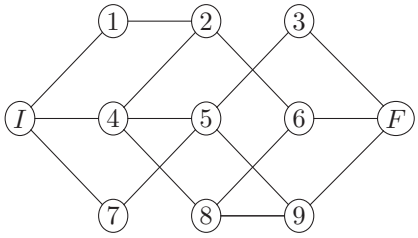

Figura 2.1: Gráfica en la que se busca ir de I a F

Iniciamos en I por lo que los candidatos son los nodos 1,4 y 7. Si elegimos a 1 sólo queda disponible 2 y así sucesivamente podemos construir la solución $S = \{(I, 1), (1, 2), (2, 6), (6, F)\}$ o alguna otra.

Otro ejemplo que podemos dar es cuando la solución se representa como una serie de números $x = (x_1, x_2, ..., x_n)$ donde cada candidato es un posible valor de las variables, mismos que dependen de las restricciones del problema, lo cual

podemos ilustrar con la siguiente instancia:

$$
\begin{aligned}
&Max\ z = 4x_1 + 2x_2 + 3x_3 + 6x_4 \\
&\text{s.a} \\
&6x_1 + 2x_2 + x_3 - 2x_4 \leq 5 \\
&4x_1 + 2x_3 + x_4 \leq 5 \\
&x_i \in \mathbb{Z}^+
\end{aligned}
$$

Por las restricciones el único valor que x_1 puede alcanzar es 0 y a x_2 podemos asignarle el valor de 0, 1 o 2, supongamos que es 2 y así sucesivamente podemos construir la solución factible (0,2,0,1).

Esta manera de construir la solución es lo que distingue a los métodos en general pero la estrategia empleada para elegir a cada candidato es lo que caracteriza a cada método constructivo y será esta estrategia la que nos permitirá determinar una buena solución (que para este ejemplo buscaría maximizar la función objetivo).

Empezaremos con el heurístico Glotón que es conocido por ser simple, ya que solamente construye una solución, y posteriormente continuaremos con el método de Colonia de Hormigas que se inspira en lo mismo que le da su nombre: el comportamiento de las hormigas.

2.1. Método Glotón

El heurístico Glotón, también llamado método Voraz, además de ser conocido por ser simple es una prueba de que un heurístico bien elegido puede obtener resultados de calidad, ya que siendo el método más básico existen algunos algoritmos basados en esta estrategia y que siempre encuentran el óptimo, por ejemplo: el algoritmo de Kruskal en el problema de árbol de peso mínimo.

Recibe el nombre de Voraz porque su estrategia consiste en tomar en cada iteración al mejor candidato para conformar la solución es decir, satura la solución con la mayor cantidad de candidatos *atractivos* basándose únicamente en la información disponible al momento y sin considerar el efecto a futuro por lo que se dice que su enfoque es miope; una vez tomada una decisión no vuelve a ser reconsiderada, lo que implica que este método nunca considera otras alternativas; por otro lado esto le da mayor rapidez.

Un ejemplo donde se toma una decisión por medio de un heurístico glotón es el siguiente: Supongamos que debemos dar la cantidad de $249 a una persona y para pagar disponemos de billetes de 100 y monedas de 10, 5, 2 y 1 y se pretende dar el menor número de monedas y billetes posible.

La estrategia voraz, en realidad resulta intuitiva: dar la mayor cantidad de billetes de 100 antes de usar las monedas, posteriormente se usarán las monedas

de 10, que serán 4 pues con 5 rebasaríamos la cantidad de 249, luego usaremos una de 5 y finalmente 4 de 1, que de hecho será la mejor decisión.

A pesar de ello el método Glotón no siempre resultará en un óptimo, la miopía mencionada anteriormente es la principal causa de falla al utilizarlo; siendo esto más notorio en algunas instancias del problema del agente viajero y del problema de cubierta de vértices, entre otros.

Las principales características de un heurístico Glotón son las siguientes:

- Se dispone de una lista de candidatos $\tilde{C}$ para formar parte de la solución.
- Se tiene una función o criterio de verificación para determinar cuando ya se ha completado una solución, la cual nos indicará cuando el método deba detenerse.
- Además de una función de selección con la que decidiremos qué candidato entra a la solución.

Como hemos comentado el heurístico devuelve la solución en forma de un conjunto; inicialmente éste será vacío pero a su término el conjunto será suficiente para identificar a una solución de la región factible. El método Glotón puede ser descrito de la siguiente manera:

Método Glotón.

1. Hacer $C = \emptyset$
2. Elegir al mejor elemento c' de $\tilde{C}$ y verificar si $C \cup c'$ es factible.
   - Si $C \cup c'$ es factible hacer $C = C \cup c'$
   - En caso contrario ir a 3
3. Verificar si C ya es una solución completa.
   - En caso de serlo terminar el procedimiento
   - En caso contrario Hacer $\tilde{C} = \tilde{C} \setminus c'$ y regresar a 2.

Como hemos mencionado la aplicación de este heurístico ha sido exitosa en algunos problemas, claro está que estos casos son problemas fáciles por lo que será necesario revisar ejemplos del heurístico Glotón aplicado a problemas que sean NP Duros.

2.1.1. Método Glotón para el problema de la mochila

En el capítulo anterior demostramos que el problema de la mochila es NP Duro, no obstante ese caso sólo es una variante del problema; de manera general el problema surge del siguiente planteamiento: Se tiene una mochila con capacidad b y n artículos cada uno con peso $c_1, c_2, ..., c_n$ los cuales aportan cierto beneficio $a_1, a_2, ..., a_n$ si son llevados en la mochila. El problema consiste en elegir la combinación de artículos que proporcione el mayor beneficio para llevarlos en la mochila por lo que el problema se formula de la siguiente manera:

$$Max\ z = \sum_{i=1}^{n} a_i x_i$$
s.a
$$\sum_{i=1}^{n} c_i x_i \leq b$$
$$x_i \in \mathbb{Z}^+ \ i = 1, 2, ..., n$$

Donde x_i es la proporción del artículo i introducida a la mochila.

Las variantes de este problema dependen de las restricciones que se tengan sobre los artículos, por como lo describimos antes debe suceder que se deben llevar una cantidad de artículos pero siendo unidades enteras, si la restricción fuera $x_i \in \{0, 1\}$ tendremos el caso en el que sólo se puede elegir una unidad de cada artículo y tuviéramos la restricción $x_i \in [0, 1]$, podemos interpretarla como la posibilidad de fraccionar el artículo pero sólo podemos llevar uno de cada uno.

Recordemos que en el capítulo anterior se demostró que el caso donde la variable x_i es binaria, es un problema difícil de resolver, por lo que podemos hacernos una idea de que el método Glotón no funcionará tan bien como el algoritmo de Kruskal para el problema de árbol de peso mínimo, sin embargo, para el caso donde la restricción es $x_i \in [0, 1]$ es posible demostrar que se encuentra siempre el óptimo tal como lo hace Brassard en [9].

En cualquiera de los casos, las soluciones pueden ser representadas como un vector en $\mathbb{R}^n$, así que los candidatos serán los posibles valores de cada variable x_i. También sabemos que no es posible rebasar la capacidad de la mochila, lo que servirá como criterio de factibilidad.

Lo único que falta por definir es lo que caracteriza al método: la elección del mejor candidato en cada iteración. Por un lado está el valor que aporta cada objeto, así que introducir primero los de mayor valor es una posibilidad; la otra es una combinación entre el peso y el beneficio: es posible calcular la aportación de cada artículo por unidad de peso mediante el cociente $\frac{a_i}{c_i}$.

Esta última manera es la que suele ser usada, ya que para los casos donde el problema no es entero sí encuentra el óptimo. El heurístico glotón para el problema de la mochila es el siguiente:

Método Glotón para el problema de la mochila.

1. Calcular el valor de la función de selección $s_i = \dfrac{a_i}{c_i}$, $i = 1, 2, 3, ..., n$ y tomar como solución inicial a todas las variables en ceros (es decir $x_i = 0$, $i = 1, 2, 3, ..., n$)
2. Sea:
   $s_{(1)} = max\{s_1, s_2, ..., s_n\}$
   $s_{(2)} = max\{\{s_1, s_2, ..., s_n\} \setminus s_{(1)}\}$
   $\vdots$

$s_{(n)} = min\{s_1, s_2, ..., s_n\}$
Es decir, ordenar los s_i de manera decreciente. Hacer $i = 1$

3. Elegir a la variable asociada a $s_{(i)}$ y darle el valor de 1.
   - Si $x = (x_1, x_2, ..., x_n) \in F_p$ mantener a la variable con el valor de 1
   - Si $x = (x_1, x_2, ..., x_n) \notin F_p$ regresar a la variable el valor de 0
4. Verificar
   - Si $i = n$ la solución final es $x = (x_1, x_2, ..., x_n)$
   - Si $i \neq n$ hacer $i = i + 1$ y regresar a 3

Esta descripción está diseñada para el problema binario y con el criterio de selección del cociente pero sus variantes son muy similares; en el caso donde la función de selección se cambia por el beneficio basta con hacer $s_i = a_i$.

Con respecto a las variantes del problema, será necesario cambiar parte del punto 3. Si tenemos la posibilidad de fraccionar los artículos, es decir, no se le da el valor de 1 sino que se le asigna el valor máximo para el cual sigue siendo factible la solución x (esto es el mínimo entre 1 y $\frac{b-(c_1x_1+c_2x_2+...+c_nx_n)}{c_i}$).

Consideremos, como ejemplo, la siguiente instancia del problema binario:

$$Max\ z = 30x_1 + 28x_2 + 50x_3 + 18x_4 + 59x_5$$
s.a
$$10x_1 + 20x_2 + 30x_3 + 40x_4 + 50x_5 \leq 100$$
$$x_i \in \{0, 1\}$$

La función de selección será la relación de beneficio por unidad de peso de manera que si seguimos el algoritmo descrito arriba tenemos lo siguiente:

Primera iteración:

Paso 1. Calcular el valor de s_i.

$$s_1 = \frac{30}{10} \quad s_4 = \frac{18}{40}$$
$$s_2 = \frac{28}{20} \quad s_5 = \frac{59}{50}$$
$$s_3 = \frac{50}{30}$$

Paso 2. Los ordenamos de manera decreciente:

$$s_{(1)} = s_1 = 3 \quad s_{(4)} = s_5 = \frac{59}{50}$$
$$s_{(2)} = s_3 = \frac{5}{3} \quad s_{(5)} = s_4 = \frac{9}{20}$$
$$s_{(3)} = s_2 = \frac{7}{5}$$

Y hacemos i = 1.

Paso 3. Seleccionamos a $s_{(1)}$ y dado que $(1, 0, 0, 0, 0)$ no sobrepasa la capacidad de la mochila la mantenemos con ese valor.

Paso 4. Como $i \neq 5$ hacemos $i = 2$ y regresamos a 3.

Segunda iteración:

Paso 3. Seleccionamos a $s_{(2)}$ y le damos valor de 1. Como $(1, 0, 1, 0, 0)$ sigue siendo factible vamos a 4.

Paso 4. Hacer $i = 3$ y regresamos a 3

Tercera iteración:

Paso 3. Seleccionamos a $s_{(3)}$ y hacemos $x_2 = 1$. Como $(1, 1, 1, 0, 0)$ el peso es de 60 sigue siendo factible y repetimos el paso 4.

Paso 4. Hacer $i = 4$ y regresamos a 3

Cuarta Iteración:

Paso 3. Continuamos con $s_{(4)}$ pero $(1, 1, 1, 0, 1)$ no es factible porque el peso es de 110 y sobrepasamos la capacidad de 100 de la mochila. De modo que $x_5 = 0$

Paso 4. Hacer $i = 5$ y vamos a 3.

Quinta Iteración:

Paso 3. Finalmente hacemos $x_4 = 1$ y dado que $(1, 1, 1, 1, 0)$ y como es factible conservamos la variable con valor de 1.

Paso 4. Como $i = 5$ hemos terminado y la solución candidata es: $s = (1, 1, 1, 1, 0)$ y $z(s) = 126$

Como hemos mencionado no se garantiza encontrar el óptimo, por lo que repetir el método con otras funciones de selección y encontrar una solución mejor no significa que hayamos realizado algo mal. Tampoco implica que ordenar los candidatos con una función específica siempre dará una solución mejor, lo que sí es posible, es implementar más de un criterio y elegir a la mejor obtenida, si es que el tiempo lo permite.

Si repetimos este mismo ejemplo con las otras dos funciones que comentamos obtenemos los resultados que podemos ver en la siguiente tabla:

Función de selección	Solución	Función objetivo
Peso	$(1, 1, 1, 1, 0)$	126
Beneficio	$(1, 0, 1, 0, 1)$	139
Peso por unidad	$(1, 1, 1, 1, 0)$	126

Ahora repetiremos el método con una versión relajada del problema, es decir, aquella donde cada variable puede tomar valores entre cero y uno. Y los resultados obtenidos son los siguientes:

Función de selección	Solución	Función objetivo
Peso	$(1, 1, 1, 1, 0)$	126
Beneficio	$(1, \frac{1}{2}, 1, 0, 1)$	153
Peso por unidad	$(1, 1, 1, 0, \frac{4}{5})$	155.2

Esto último es sólo para comprobar que la función de peso por unidad de beneficio sí encuentra la mejor solución, en este caso particular podemos verificar que la solución $(1, 1, 1, 0, \frac{4}{5})$ es óptima.

2.1.2. Método Glotón para el problema de número cromático

Recordemos que el problema del número cromático consiste en colorear una gráfica con el mínimo número de colores de manera que cada nodo tenga un color diferente al de sus nodos adyacentes. Podemos representar una solución como un conjunto de parejas (x, y) que representan a un nodo y el color que le asignamos, por ejemplo la solución $\{(1, 1), (2, 2), ..., (n, n)\}$ que consiste en colorear a cada nodo de un color distinto para una gráfica con n nodos.

El razonamiento glotón que usaremos en este caso es el siguiente: El objetivo es asignar al candidato más atractivo y siendo los colores los que debemos asignar podemos deducir que éstos son los candidatos. En la primera iteración no hay un favorito, ya que es indistinto el color que asignemos, sin embargo, una vez que le asignemos uno al primero nodo, digamos $(1, 1)$, el color 1 se volverá el candidato más atractivo pues si queremos minimizar el número de colores utilizados lo ideal es reutilizarlo.

En caso de no poder utilizarlo elegimos alguno de los no utilizados, con lo que tendremos otro candidato igual más atractivo. Siguiendo esta secuencia debemos terminar de colorear la gráfica con la menor cantidad de colores.

Otra manera de seguir el procedimiento consiste en probar cada color en todos los nodos, pues si utilizamos un color ya sabemos que es el candidato más atractivo. Dada una gráfica $G = (X, A)$ necesitamos que el conjunto de nodos X esté numerado de alguna manera $X = \{v_1, v_2, ..., v_n\}$ ya que aprovecharemos esta numeración; lo que seguiría es colorear el nodo v_1 de un color, revisar si el nodo v_2 puede ser coloreado del mismo y en caso de ser así hacerlo, lo mismo con el nodo v_3 verificando que no sea adyacente a v_1 ni a v_2, así repetimos el proceso

hasta verificar si v_n puede o no ser coloreado. Al terminar se elige al siguiente nodo no coloreado, por ejemplo v_3 si v_1 y v_2 ya lo estuvieran, y repetimos la misma verificación; seguiríamos así hasta colorear todos los vértices.

De manera más específica el método es el siguiente:

Método Glotón para el problema del número cromático.

1. Elegir de manera aleatoria un nodo v_1 y numerar el resto como $v_2, v_3, ..., v_n$. Hacer $N_c = \emptyset$, $C = \emptyset$, $X' = \{v_1, v_2, ..., v_n\}$, k = 1 y j = 1
2. Verificar si v_j puede colorearse con el color k.
   - Si es posible entonces hacer $N_c = N_c \cup \{v_j\}$, $C = C \cup \{k\}$, $X' = X' \setminus \{v_j\}$
   - En caso contrario ir a 3. No se pueden colorear los nodos ya coloreados ni los que tengan vecinos coloreados con el color k
3. Verificar si j = n
   - Si es igual a n hacer k = k+1 y j= 1
   - En caso contrario hacer j = j+1
4. Calcular $|X'|$
   - Si es igual a cero terminar. Todos los nodos han sido coloreados
   - Si $|X'| \neq 0$ Regresar a 2

En este problema el método falla al depender de la numeración de los nodos; pongamos de ejemplo la figura 2.2:

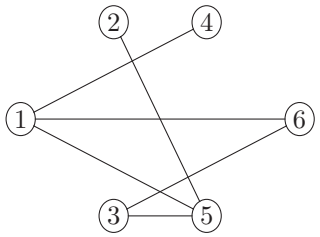

Figura 2.2: Ejemplo para el problema de número cromático

Primera Iteración:

Paso 1. Ya que los nodos están numerados haremos $v_i = i$ es decir, aprovecharemos esa numeración. Aprovecharemos el orden que ya aparece ahí. Sea $N_c = \emptyset$, $C = \emptyset$, $X' = \{1, 2, 3, 4, 5, 6\}$, k=1 y j=1

Paso 2. Al ser el primer nodo a colorear v_1 puede ser coloreado de color 1, de modo que $N_c = \{v_1\}$, $C = \{1\}$ y $X' = \{2, 3, 4, 5, 6\}$

Paso 3. Hacemos $j = 2$ y pasamos a 4.

Paso 4. Como $X' \neq \emptyset$ regresamos a 2.

Segunda Iteración: *Paso 2.* Como v_2 no es vecino de v_1 también puede colorearse de 1. $N_c = \{v_1, v_2\}$, $C = \{1\}$ y $X' = \{3, 4, 5, 6\}$ *Paso 3.* Ahora $j = 3$ y vamos a 4. *Paso 4.* Regresamos a 2.

Tercera Iteración: *Paso 2.* El nodo v_3 no es adyacente a v_1 y v_2 de modo que también puede colorearse de 1. $N_c = \{v_1, v_2, v_3\}$, $C = \{1\}$ y $X' = \{4, 5, 6\}$ *Paso 3.* Ahora hacemos $j = 4$. *Paso 4.* Como $|X'| = 3$ regresamos a 2.

Cuarta Iteración: *Paso 2.* v_4 no puede colorearse de 1 porque es adyacente a v_1, por lo que los conjuntos no cambian en esta iteración. *Paso 3.* $j = 5$ y pasamos a 4. *Paso 4.* Como no hubo cambios en X' seguimos iterando

Quinta Iteración: *Paso 2.* Tampoco v_5 puede colorearse de 1 porque es vecino de 2 y 3. *Paso 3.* Hacemos $j = 6$ *Paso 4.* Nuevamente tenemos que iterar debido a que $|X'|$ sigue siendo 3.

Sexta Iteración: *Paso 2.* v_6 no se colorea de 1 porque es adyacente a 1 y 3. *Paso 3.* Como ya revisamos todos los nodos y solo los primeros 3 se pudieron colorear de 1 probaremos con un segundo color. Hacemos $k = 1$ y $j = 1$ para reanudar la búsqueda. *Paso 4.* Repetimos otra iteración para tratar de colorear los demás nodos.

El paso 3 indica que los nodos coloreados no pueden volverse a colorear de manera que podemos saltarnos las iteraciones para v_1, v_2 y v_3 y hacer $j = 4$

Séptima Iteración: *Paso 2.* v_4 puede colorearse de 2 porque es el primero de este color. De modo que $N_c = \{v_1, v_2, v_3, v_4\}$, $C = \{1, 2\}$ y $X' = \{5, 6\}$ *Paso 3.* Hacemos $j = 5$ y pasamos a 4. *Paso 4.* Aún no sucede que $|X'| = 0$ por lo que iteramos de nuevo

Octava Iteración: *Paso 2.* El nodo v_5 no es vecino de v_4 por lo que sí puede colorearse de 2. Hacemos $N_c = \{v_1, v_2, v_3, v_4, v_5\}$, $C = \{1, 2\}$ y $X' = \{6\}$ *Paso 3.* Hacemos $j = 6$ *Paso 4.* Aun falta un nodo por ser coloreado de modo que regresamos a 2

Novena Iteración: *Paso 2.* El nodo v_6 no tiene adyacencias con v_4 y v_5 así que también es coloreado de 2. $N_c = \{v_1, v_2, v_3, v_4, v_6\}$, $C = \{1, 2\}$ y $X' = \emptyset$ *Paso 3.* Hemos terminado por lo que hacemos $k = 1$ y $j = 1$ *Paso 4.* Finalmente hemos terminado de colorear la gráfica. Se necesitaron 2 colores.

En resumen, al realizar el método tendremos que los nodos 1,2 y 3 pueden ser coloreados igual, pero el 4 ya no, por lo que elegimos otro color; posteriormente los nodos 5 y 6 pueden ser del mismo color de 4, es decir, la gráfica puede ser coloreada con solo 2.

Ahora analicemos la gráfica de la figura 2.3:

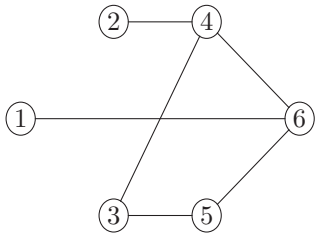

Figura 2.3: Segundo ejemplo para el problema del número cromático

En este caso los nodos 1, 2 y 3 pueden ser del mismo, 4 y 5 también, pero el nodo 6 debe ser coloreado diferente por lo que se necesitaron 3 colores. Podemos re ordenar la gráfica de la siguiente manera:

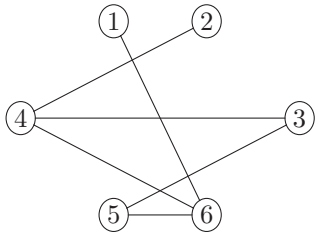

Figura 2.4: Gráfica reordenada

Esperamos que así sea más fácil notar que es la misma gráfica que resolvimos primero, la diferencia está en la numeración de los vértices, que como dijimos, es donde se ubica el problema. No existe alguna manera de ordenar los vértices que garantice que se encuentre el óptimo. Lo que sí tenemos es la garantía de que el óptimo sí puede ser encontrado con este método, lo que significa que existe algún ordenamiento de los nodos tal que se encontrará una solución óptima, demostración que también realiza Brassard y podemos consultar en [9].

2.1.3. Método Glotón para el problema del agente viajero

El problema del agente viajero es el caso donde es más evidente la miopía del método. Para este problema podemos utilizar como candidatos a los nodos, situándonos en un nodo inicial y avanzar posteriormente al más cercano, sin embargo, estas elecciones pueden ocasionar que más adelante los nodos que falten por visitar sólo puedan ser alcanzados por medio de aristas con distancias muy grandes. A continuación describiremos el método para construir la solución por el método Glotón.

Método Glotón para el problema del Agente Viajero. Se inicia con una gráfica $G = (X, A)$. Y con los nodos numerados como $1, 2, 3, ..., n$

1. Iniciamos el recorrido en un nodo seleccionado al azar, supongamos que es m. Sea $i = m$, $V = \{m\}$ y sea $S = \emptyset$

2. Sea $N = \{j \in X \setminus V : (i,j) \in A\}$ el conjunto de vecinos de i no visitados aún. Sea $\tilde{i}$ el nodo más cercano a i (es decir, $d_{ij} \leq d_{ij'} \forall j' \in N$) , entonces hacer $S = S \cup (i, \tilde{i})$ y $V = V \cup \tilde{i}$.
3. Verificar si $|V| = |X|$
   - Si esto ocurre ir a 4.
   - En caso contrario hacer $i = \tilde{i}$ y regresar a 2.
4. Añadir la arista $(\tilde{i}, m) a S$, una vez realizado esto S es el ciclo hamiltoniano encontrado por la hormiga.

La miopía del método es más notoria aquí, observemos que en el ejemplo de la figura 2.5 no solo no encontramos el óptimo, al contrario, encontramos la peor solución posible:

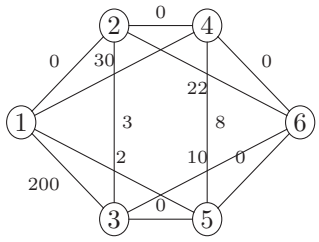

Figura 2.5: Ejemplo para el problema del agente viajero

Primera Iteración

Paso 1. Seleccionemos el nodo 1 como inicial. de modo que $i = 1$, $V = \{1\}$ y $S = \emptyset$

Paso 2. $N = \{2, 3, 4, 5\}$ de entre los cuales el más cercano es 2 ya que la distancia entre ellos es 0. Por lo que $S = \{(1, 2)\}$ y $V = \{1, 2\}$

Paso 3. $|V| = 1$ por lo que $i = 2$ y regresamos a 2

Segunda Iteración

Paso 2. $N = \{3, 4, 6\}$ por lo que elegimos a 4 como vecino más cercano. $i = 4$, $V = \{1, 2, 4\}$ y $S = \{(1, 2), (2, 4)\}$

Paso 3. $|V| < 6$ de modo que $i = 4$ y volvemos a 2

Tercera Iteración

Paso 2. $N = \{5, 6\}$ así que el vecino más cercano es 6. Sea $i = 6$, $S = \{(1, 2)(2, 4)(4, 6)\}$ y $V = \{1, 2, 4, 6\}$

Paso 3. $|V| = 3$ así que hacemos $i = 6$ y regresamos a 2

Cuarta Iteración

Paso 2. $N = \{3, 5\}$ de los cuales el más cercano es 5. Ahora hacemos $i = 5$, $S = \{(1, 2)(2, 4)(4, 6), (6, 5)\}$ y $V = \{1, 2, 4, 6, 5\}$

Paso 3. $|V| = 5$ lo que significa que en una iteración más terminamos. Sea $i = 5$ y pasamos a 2

Quinta Iteración

Paso 2. $N = \{3\}$ por lo que 3 es el siguiente nodo visitado. $i = 3$, $S = \{(1, 2)(2, 4)$ $(4, 6), (6, 5), (5, 3)\}$ y $V = \{1, 2, 4, 6, 5, 3\}$

Paso 3. Finalmente $|V| = 6$ por lo que pasamos al paso 4.

Paso 4. Añadimos la arista $(1, 3)$ a S. De esta manera queda la solución: $S = \{(1, 2)(2, 4)\ (4, 6), (6, 5), (5, 3), (3, 1)\}$

Resulta que esta solución tiene una distancia de 200 porque para cerrar el ciclo teníamos como única opción añadir la arista con mayor distancia de toda la red. No es que hayamos implementado el método de manera inadecuada, simplemente ha ocurrido algo que supuestamente no debería suceder muy a menudo; podríamos decir que el heurístico glotón no es conveniente de aplicar cuando una de las aristas tiene un peso demasiado diferente al de los demás.

Por último debemos mencionar que estas instrucciones no funcionan para todas las instancias del agente viajero, debemos limitar su uso a problemas donde cada nodo sea adyacente a los demás, pues en otro caso no se garantiza encontrar una solución. Por ejemplo, si retiramos la arista (1,3) no se encontrará una solución pese que a que sí existe alguna como recorrer los nodos 1,2,3,6,4,5 y 1.

2.2. Colonia de Hormigas

Como su nombre lo indica este método fue inspirado en el comportamiento de las hormigas, específicamente en la estrategia empleada al recolectar su comida. Estos insectos siguen una ruta desde el hormiguero hasta la fuente de alimento y todas siguen el mismo camino que generalmente resulta ser el más corto, lo cual es interesante por la técnica empleada para encontrar esta ruta, pues las hormigas son prácticamente ciegas, así que no se valen de técnicas visuales.

Al encontrarse en esta situación las hormigas recurren a otro tipo de señales, al caminar las hormigas dejan un rastro de *feromonas* que pueden seguir para regresar al hormiguero mismo que las otras pueden seguir también. De esta ma-

nera, cuando una encuentra comida las demás pueden seguir el rastro y ayudar en su traslado.

Sin embargo, si las hormigas siguieran cualquier rastro de feromonas esta técnica sería un completo desastre, por lo que ellas no siempre siguen los rastros encontrados, la elección se hace con cierta probabilidad misma que depende de la fuerza que tenga, así si una hormiga encuentra alimento al regresar al hormiguero depositará más feromonas por lo que el rastro será mas fuerte y será mas probable que otras lo sigan haciéndolo más fuerte aún.

De esta misma manera es como encuentran la ruta más corta: imaginemos que hay dos caminos a una misma fuente de alimento y que el primero es más corto que el segundo, aún cuando varias hormigas elijan ambas rutas aquellas que recorran el primero llegarán antes al hormiguero por lo que pasarán más veces por el mismo camino haciendo más fuerte el rastro lo que finalmente hará más probable la elección del camino 1.

Estos rastros no son permanentes, con el paso del tiempo la feromona se evapora para prevenir que las hormigas sigan el rastro a fuentes de alimento que ya se han terminado además de evaporar rastros que son poco usados; todo el proceso en conjunto permite que quede un único rastro que es el más corto.

La idea general del heurístico es la misma, enviar varias *hormigas* a explorar las posibles rutas imitando la técnica de las feromonas hasta encontrar una buena. Esto implica que un problema que se quiera resolver por este método debe poder ser representado por una red.

De manera general el heurístico de Colonia de Hormigas es el siguiente:

Método de Colonia de Hormigas.
Se realiza un número T^{max} de iteraciones, en cada una n hormigas construyen una solución completa.

1. Hacer i=1, t=1 y $C_i = \emptyset$
2. Añadir de manera aleatoria un candidato a C_i
3. Verificar
   - Si C_i es una solución completa ir a 4
   - En caso contrario regresar a 2
4. Actualizar de manera local el rastro de feromonas.
5. Verificar
   - Si $i = n$ ir a 6
   - Si $i < n$ hacer i=i+1 e ir a 2
6. Actualizar de manera global los caminos recorridos por las hormigas
7. Verificar el número de iteraciones
   - Si $t = T^{max}$ terminar
   - Si $t < T^{max}$ hacer t=t+1, i=1 e ir a 2

A pesar de conocerse como método de Colonia de Hormigas a cualquiera que siga estas instrucciones, se han desarrollado variantes que pretenden obtener mejores desempeños, mencionaremos el Sistema de Hormigas (Ant System), el Sistema de Colonia de Hormigas (Ant Colony System) y el Sistema de Hormigas Max Min (Max Min Ant System).

El **Sistema de Hormigas** es el más básico, como hemos mencionado está pensado para realizarse sobre problemas que puedan ser representados en una gráfica en la cual se sitúan las n hormigas distribuidas en los nodos, el siguiente que se elige es algún vecino según la "feromona"que tenga la arista que los une. La elección se lleva a cabo de manera probabilística usando la siguiente fórmula:

$$P_k(i,j) = \begin{cases} \frac{\tau(i,j)^\alpha \eta(i,j)^\beta}{\sum_{u \in N_{k,i}} \tau(i,u)^\alpha \eta(i,u)^\beta} & N_{k,i} := \{u : u \in X \text{y es válido recorrer la arista (i,u)}\} \\ 0 & \text{En otro caso} \end{cases}$$

$P_k(i,j)$ denota la probabilidad de que la k-ésima hormiga, estando en el vértice i se mueva al vértice j, esta probabilidad está determinada por dos elementos principales: $\tau(i,j)$ representa el rastro de feromonas que tiene la arista (i,j) y $\eta(i,j)$ es un valor asociado al atractivo de incluir la arista (i,j) en la solución, lo que podríamos considerar como información heurística, por citar un ejemplo se puede considerar el recíproco del costo de incluir la arista $\frac{1}{c_{ij}}$.

Al inicio del método el valor de $\tau(i,j)$ es igual a un valor pequeño y generalmente constante para todos las aristas denotado por τ_0. Los valores α y β son parámetros para determinar la importancia de $\tau(i,j)$ con respecto a $\eta(i,j)$. Si $\alpha = 0$ significaría que la feromona no está siendo tomada en cuenta y sólo usamos la información heurística, en cuyo caso el método se asemeja bastante al glotón, ya que las aristas con mayor beneficio tendrán mayor probabilidad de ser elegidos; y si $\beta = 0$ ocurre lo contrario, la información heurística no se considera para nada, la elección está basada en las feromonas, la desventaja de esto es que se puede producir un estancamiento del método al caer en un óptimo local.

Este método no considera una actualización local, una vez que todas las hormigas han terminado su recorrido se procede a hacer la actualización global del rastro de la siguiente manera:

$$\tau(i,j) = (1-\rho)\tau(i,j) + \sum_{k=1}^{m} \Delta\tau_k(i,j)$$

La actualización consta de los dos procesos que suceden con las hormigas, el primero es la evaporación del rastro ($\rho \in (0,1)$ es la tasa de evaporación) y el segundo es el depósito que hacen las hormigas al "pasar"por la arista, $\Delta\tau_k(i,j)$ corresponde al incremento en el rastro debido al paso de la k-ésima hormiga en esa arista. La cantidad de feromona debe estar en función de la calidad de la solución encontrada, el plan es que si la solución es mala (si fuera un problema de

ruta más corta, una solución mala sería una solución con una distancia grande) el rastro sea menor. Esto significa que $\Delta\tau_k(i,j) = f(c(S_k))$ con S_k la solución encontrada por la hormiga k. Suele usarse la siguiente fórmula:

$$\Delta\tau_k(i,j)\begin{cases}\frac{1}{c(S_k)} & (i,j)\in S_k\\ 0 & \text{En otro caso}\end{cases}$$

Cabe mencionar que se puede cambiar por otra función, por ejemplo multiplicar por una constante Q para hacer $\frac{Q}{c(S_k)}$ o utilizar $\frac{1}{c_{ij}}$.

Una variante de este mismo método es el Colonia de Hormigas *Elitista* que hace una actualización especial. Si $\tilde{S}$ es la mejor solución encontrada en la iteración por la hormiga $\tilde{k}$, se actualizan las aristas:

$$\tau(i,j) = \tau(i,j) + \rho e \Delta\tau_{\tilde{k}}(i,j)$$

Este rastro adicional quiere representar el paso de e hormigas elitistas por este camino, de manera que e es un número fijado al inicio del método.

Posteriormente se desarrolló una variante de Sistema de Hormigas, el **Sistema de Colonia de Hormigas** que difiere de este en 3 aspectos:

Primero hay una variante en la selección de la ruta. Se genera un número aleatorio p_0 y definimos un parámetro p que será constante, dependiendo de el valor de p_0 elegimos una de las siguientes fórmulas:

- Si $p_0 \leq p$ hacemos:

$$P_k(i,j) = \begin{cases}1 & \text{Si } j = argmax\{\tau(i,j)\eta(i,j)^\beta\}\\ 0 & \text{En otro caso}\end{cases}$$

- Si $p_0 > p$:

$$P_k(i,j) = \begin{cases}\dfrac{\tau(i,j)^\alpha\eta(i,j)^\beta}{\displaystyle\sum_{u\in N_{k,i}}\tau(i,u)^\alpha\eta(i,u)^\beta} & siu \in N_{k,i}\\ 0 & \text{En otro caso}\end{cases}$$

Esto significa que con probabilidad p elegiremos la mejor opción, con base en la información que disponemos hasta este momento. Y con probabilidad $(1-p)$ elegiremos la siguiente arista de la misma manera que en Sistema de Hormigas.

La segunda diferencia es que aquí sí hay una actualización local de las feromonas, al termino del recorrido cada hormiga actualiza las aristas por donde pasó de la siguiente manera:

$$\tau(i,j) = (1-\rho)\tau(i,j) + \rho\tau_0$$

Donde τ_0 es el mismo valor inicial con que marcamos las aristas al inicio del método.

Por último, la tercera diferencia que tienen es en el proceso de actualización global, pues en esta etapa Sistema de Colonia de Hormigas sólo actualiza a la mejor solución de la iteración y se hace de la siguiente manera:

$$\tau(i,j) = \tau(i,j) + \rho\Delta\tau_{\tilde{k}}(i,j)$$

La última variante que comentaremos es el **Sistema de Hormigas Max Min** que también es una variante del Sistema de Hormigas, esta variante pretende aprovechar mejor la información obtenida de las soluciones buenas y evitar la convergencia prematura a un óptimo local.

La actualización global se hace evaporando los rastros de manera usual al hacer $(1-\rho)\tau(i,j)$ lo que cambia es la actualización del rastro de feromonas, pues nuevamente se actualiza sólo la mejor solución (ya sea la mejor de la iteración o la mejor global), es decir:

$$\tau(i,j) = \tau(i,j) + \Delta\tau_{\tilde{k}}(i,j)\ \forall a_{ij} \in S_{\tilde{k}}$$

Para prevenir la convergencia prematura esta variante sugiere definir cotas para el valor de τ: τ_{min} y τ_{max}, de esta manera se garantiza que todos las aristas pueden ser visitados con una probabilidad aceptable y ninguno tendrá una probabilidad muy elevada de ser elegido.

Sobre la cota superior se sugiere calcularla como:

$$\tau_{max} = \frac{1}{f(c(S^*))}$$

Con S^* la solución óptima del problema. Esa función puede ser la misma que la ocupada para calcular Δ. A falta de la solución óptima podemos sustituirla por la mejor solución global encontrada hasta el momento (como la solución encontrada por un Glotón). Con respecto a la cota inferior basta con que sea una constante menor que la cota superior.

La última diferencia con las demás variantes es que Max Min no asigna τ_0 como valor inicial, sino el valor máximo que puede tener (τ_{max}), de esta manera hay más diversificación en las primeras iteraciones pues las diferencias serán menores que con valores pequeños.

A continuación presentaremos dos ejemplos de estos métodos, primero para el problema del agente viajero y posteriormente para le problema de clique de cardinalidad máxima.

2.2.1. Colonia de Hormigas para el problema del agente viajero

Recordemos que este problema tiene como finalidad encontrar el ciclo hamiltoniano de menor "distancia" en una red, en otras palabras buscamos una ruta que pase por todos los nodos de la gráfica y que sea la más corta lo cual es perfecto para probar el funcionamiento de este método. Las hormigas pueden ser iniciadas en un nodo y avanzar a sus vecinos siempre y cuando no se llegue a un nodo visitado anteriormente.

Para resolver este problema en una gráfica $G = (X, A)$ usaremos el sistema de hormigas implementado de la manera siguiente:

Método de Colonia de Hormigas para el problema del agente viajero.

1. Distribuir las n hormigas en los nodos de X. Se sugiere hacer $n = |X|$ para colocar una por nodo.
2. Hacer t = 0 y $\tau(i,j) = \tau_0 \forall (i,j) \in A$
3. Construir una solución para cada una de las n hormigas de la siguiente manera.
   - Cada hormiga inicia en un nodo v. La elección del nodo a seguir se hace según el rastro de feromona que posean sus aristas vecinas siempre que no se llegue a un nodo visitado anteriormente excepto cuando se vuelve a llegar a v para completar el ciclo. Calcular la probabilidad se calcula como:

$$P_k(i,j) = \begin{cases} \frac{\tau(i,j)^\alpha \eta(i,j)^\beta}{\sum_{u \in N_{k,i}} \tau(i,u)^\alpha \eta(i,u)^\beta} & \text{si } u \in N_{k,i} \\ 0 & \text{En otro caso} \end{cases}$$

4. Actualizar el rastro de feromonas con la fórmula descrita anteriormente.

$$\tau(i,j) = (1-\rho)\tau(i,j) + \sum_{k=1}^{m} \Delta\tau_k(i,j)$$

5. Hacer $t = t + 1$
   - Si $t < T^{max}$ regresar a 2
   - En caso contrario terminar

La implementación de este método es un poco más pesada que el resto por el número de soluciones que se construyen, supongamos que tenemos que encontrar el ciclo hamiltoniano más corto de la figura 2.6:

Las distancias de las aristas son las siguientes:

$$\begin{array}{lll} (1,2) = 10 & (2,3) = 20 & (3,5) = 60 \\ (1,3) = 30 & (2,4) = 80 & (3,6) = 4 \\ (1,4) = 4 & (2,5) = 4 & (4,5) = 13 \\ (1,5) = 70 & (2,6) = 50 & (4,6) = 40 \\ (1,6) = 44 & (3,4) = 24 & (5,6) = 111 \end{array}$$

Como tiene 6 nodos la sugerencia es utilizar 6 hormigas y por probar un número, hagamos 6 iteraciones. Esto implica que el heurístico construirá 36 soluciones, cada una paso a paso.

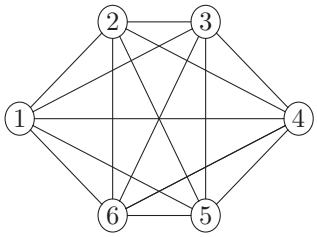

Figura 2.6: Ejemplo para el problema del agente viajero usando colonia de hormigas

Por esta razón no explicaremos cada una de las iteraciones de este método. Realizaremos la construcción de una solución por parte de una hormiga y el resto solo las indicaremos. De la misma manera, la primera actualización del rastro de feromonas será paso a paso y las siguientes serán más resumidas[1]. Pero antes, detallaremos un poco más las instrucciones para construir una solución:

Instrucciones para la construcción de un ciclo hamiltoniano por medio de la hormiga m.
Como hemos dicho que se utilizará el mismo número de hormigas que el número de nodos en la gráfica haremos que la m-ésima hormiga inicie su recorrido en el nodo m.

1. La hormiga inicia su recorrido en el nodo m. Sea $i = m$ y $V = \{m\}$ (Este será el conjunto que contiene a todos los nodos ya visitados) y sea $C_m = \emptyset$
2. Elegimos al siguiente nodo a visitar entre los vecinos de i que no pertenezcan a V (llamaremos a este conjunto $N_{k,i}$) con la siguiente probabilidad:

$$P_k(i,j) = \begin{cases} \dfrac{\tau(i,j)^{\alpha}\eta(i,j)^{\beta}}{\sum_{u \in N_{k,i}} \tau(i,u)^{\alpha}\eta(i,u)^{\beta}} & si\ u \in N_{k,i} \\ 0 & \text{En otro caso} \end{cases}$$

3. Sea $\tilde{i}$ el nodo elegido, entonces hacer $S_m = S_{\cup}(i, \tilde{i})$ y $V = V \cup \tilde{i}$.
4. Verificar si $|C_m| = |X| - 1$
   - Si esto ocurre ir a 5.
   - En caso contrario hacer $i = \tilde{i}$ y regresar a 2.
5. Añadir la arista $(\tilde{i}, m)$ a C_m, una vez realizado esto C_m es el ciclo hamiltoniano encontrado por la hormiga.

Con estas especificaciones construiremos las soluciones con cada hormiga. Cabe mencionar que la elección aleatoria del siguiente vecino se realizará de la siguiente manera:

Sean $j_1, j_2, j_3, ..., j_r$ los vecinos que podemos elegir desde el nodo i cada uno con probabilidad $P_k(i, j_n)$, $n = 1, 2, ..., r$. Sabemos que $\sum_{n=1}^{r} P_k(i, j_n) = 1$ de modo que podemos generar un número aleatorio p entre 0 y 1 y elegir al nodo s si

[1]Para consultar la implementación completa se puede utilizar el código diseñado para este problema.

$p \in (\sum_{n=1}^{r} P_k(i, j_n - 1), \sum_{n=1}^{r} P_k(i, j_n)]$.

Ya con todas las instrucciones procederemos con el ejemplo: Como mencionamos trabajaremos con 6 hormigas y repetiremos el procesos también 6 veces, los parámetros que necesitamos son $\alpha = 1$, $\beta = 1$, $\tau_0 = \frac{1}{10} = 0.1$ y $\rho = 0.22$.

Primera Iteración

Paso 1. Cada hormiga será iniciada en un nodo diferente del 1 al 6

Paso 2. Iniciamos el rastro de las aristas con t_0, es decir, $\tau(i,j) = \tau_0 \forall (i,j) \in A$ y sea t =0

Paso 3. Construimos una solución para cada hormiga.

- Primera hormiga

1. Inicia en el nodo 1, sea $i = 1$, $V = \{1\}$ y $C_1 = \emptyset$
2. Los vecinos disponibles son $N_{1,1} = \{2, 33, 4, 5, 6\}$. Calculamos las probabilidades:

j	$\tau(1,j) * \eta(1,j)$	$P_1(1,j)$
2	$0.1 * \frac{1}{10} = \frac{1}{100}$	$\frac{231}{971}$
3	$0.1 * \frac{1}{30} = \frac{1}{300}$	$\frac{77}{971}$
4	$0.1 * \frac{1}{4} = \frac{1}{40}$	$\frac{1155}{1942}$
5	$0.1 * \frac{1}{70} = \frac{1}{700}$	$\frac{33}{971}$
6	$0.1 * \frac{1}{44} = \frac{1}{440}$	$\frac{105}{1942}$
	suma $= \frac{971}{23100}$	

   Generamos un número aleatorio $r = 0.365826$ y elegimos el nodo según el intervalo donde ha caído r

   - Si $r \in [0, 0.237899]$ elegimos a 2
   - Si $r \in (0.237899, 0.317198]$ elegimos a 3
   - Si $r \in (0.317198, 0.911946]$ elegimos a 4
   - Si $r \in (0.911946, 0.945932]$ elegimos a 5
   - Si $r \in (0.945932, 1]$ elegimos a 6

3. Como ocurrió la tercera opción elegimos como siguiente nodo a visitar a 4. De modo que $C_1 = \{(1,4)\}$ y $V = \{1,4\}$
4. Como $|C_1| = 1$ no tenemos el ciclo, así que debemos buscar otro nodo a donde ir (hacemos $i = 4$ y regresamos a 2)

2. $N_{1,4} = \{2, 3, 5, 6\}$ son los nodos no visitados aún. Volvemos a calcular las probabilidades.

j	$\tau(4,j) * \eta(4,j)$	$P_1(4,j)$
2	$0.1 * \frac{1}{80} = \frac{1}{800}$	$\frac{39}{487}$
3	$0.1 * \frac{1}{24} = \frac{1}{240}$	$\frac{130}{487}$
5	$0.1 * \frac{1}{13} = \frac{1}{130}$	$\frac{240}{487}$
6	$0.1 * \frac{1}{40} = \frac{1}{400}$	$\frac{78}{487}$
	suma $= \frac{487}{31200}$	

El nuevo valor de r es 0.5339936, de modo que el nodo 5 es el que elegimos ahora.

3. Como el siguiente nodo a visitar es 5 hacemos $C_1 = \{(1,4),(4,5)\}$ y $V = \{1,4,5\}$
4. Aun no tenemos un ciclo pues $|C_1| = 2$ de modo que volvemos a 2 (con $i = 5$)
2. Los nuevos candidatos son $N_{1,5} = \{2,3,6\}$. Volvemos a calcular las probabilidades:

j	$\tau(5,j) * \eta(5,j)$	$P_1(5,j)$
2	$0.1 * \frac{1}{4} = \frac{1}{40}$	$\frac{165}{236}$
3	$0.1 * \frac{1}{60} = \frac{1}{600}$	$\frac{11}{236}$
6	$0.1 * \frac{1}{11} = \frac{1}{110}$	$\frac{15}{59}$
	suma $= \frac{59}{1650}$	

Ahora $r = 0.4171116$ de modo que nos quedamos con 2

3. $C_1 = \{(1,4),(4,5),(5,2)\}$ y $V = \{1,4,5,2\}$
4. El ciclo sigue sin completarse $|C_1| = 3$ de modo que hacemos $i = 2$ y regresamos a 2.
2. En este momento ya sólo quedan dos nodos sin visitar $N_{1,2} = \{3,6\}$ así que las probabilidades para elegir son:

j	$\tau(2,j) * \eta(2,j)$	$P_1(2,j)$
3	$0.1 * \frac{1}{20} = \frac{1}{200}$	$\frac{5}{7}$
6	$0.1 * \frac{1}{50} = \frac{1}{500}$	$\frac{2}{7}$
	suma $= \frac{7}{100}$	

y $r = 0.9322815$ por lo que elegimos a 6

3. Tenemos $C_1 = \{(1,4),(4,5),(5,2),(2,6)\}$ y $V = \{1,4,5,2,6\}$
4. Esta vez hemos visitado todos los nodos de modo que pasamos a 5
5. Añadimos la arista $(3,1)$ a C_1 para tener el ciclo:

$$C_1 = \{(1,4),(4,5),(5,2),(2,6),(6,3),(3,1)\}$$

Con una distancia total de 105 unidades

- Segunda hormiga

Siguiendo los mismos pasos que con la primera se obtiene la solución:

$$C_2 = \{(2,5),(5,4),(4,3),(3,6),(6,1),(1,2)\}$$

Con una distancia de 99 unidades

- Tercera hormiga

  Siguiendo los mismos pasos que con la primera se obtiene la solución:

  $$C_3 = \{(3,6),(6,4),(4,1),(1,2),(2,5),(5,3)\}$$

  Con una distancia de 122 unidades

- Cuarta hormiga

  Siguiendo los mismos pasos que con la primera se obtiene la solución:

  $$C_4 = \{(4,3),(3,2),(2,5),(5,6),(6,1),(1,4)\}$$

  Con una distancia de 107 unidades

- Quinta hormiga

  Siguiendo los mismos pasos que con la primera se obtiene la solución:

  $$C_5 = \{(5,2),(2,1),(1,4),(4,3),(3,6),(6,5)\}$$

  Con una distancia de 57 unidades

- Sexta hormiga

  Siguiendo los mismos pasos que con la primera se obtiene la solución:

  $$C_6 = \{(6,3),(3,1),(1,2),(2,5),(5,4),(4,6)\}$$

  Con una distancia de 101 unidades

Terminamos con la construcción de soluciones (Paso 3)

Paso 4. Actualizamos el rastro de feromonas.

Sobre la actualización del rastro sabemos que la evaporación en cada arista es $\tau(i,j) = \tau(i,j) * (1-\rho)$ y es para todas. Por otro lado $\Delta\tau_k(i,j) = \frac{1}{c(S_m)}$, de modo que iremos revisando por donde pasó cada hormiga para hacer la actualización.

- Evaporación

$\tau(1,2) = \frac{1}{10} * (1-0.22) = 0.078$ $\quad$ $\tau(2,6) = \frac{1}{50} * (1-0.22) = 0.0156$
$\tau(1,3) = \frac{1}{30} * (1-0.22) = 0.026$ $\quad$ $\tau(3,4) = \frac{1}{24} * (1-0.22) = 0.0325$
$\tau(1,4) = \frac{1}{4} * (1-0.22) = 0.195$ $\quad$ $\tau(3,5) = \frac{1}{60} * (1-0.22) = 0.013$
$\tau(1,5) = \frac{1}{70} * (1-0.22) = 0.0111$ $\quad$ $\tau(3,6) = \frac{1}{4} * (1-0.22) = 0.0195$
$\tau(1,6) = \frac{1}{44} * (1-0.22) = 0.0117$ $\quad$ $\tau(4,5) = \frac{1}{13} * (1-0.22) = 0.06$
$\tau(2,3) = \frac{1}{20} * (1-0.22) = 0.039$ $\quad$ $\tau(4,6) = \frac{1}{40} * (1-0.22) = 0.0195$
$\tau(2,4) = \frac{1}{80} * (1-0.22) = 0.00975$ $\quad$ $\tau(5,6) = \frac{1}{11} * (1-0.22) = 0.0700909$
$\tau(2,5) = \frac{1}{4} * (1-0.22) = 0.195$

- Actualización hormiga 1 Como la hormiga 1 encontró un camino de 105 unidades de distancia el rastro que se depositará en cada arista será de $\frac{1}{105} \approx 0.00952$
- Actualización hormiga 2 La segunda hormiga recorrió todos los nodos con una distancia de 99 unidades, por lo que su rastro es aproximadamente de 0.010101
- Actualización hormiga 3 La tercera hormiga encontró el camino más largo en esta iteración de modo que su rastro de feromonas será menor que las demás (0.008196)
- Actualización hormiga 4 La cuarta hormiga depositará un aproximado de 0.009345 ya que su recorrido fue de 107 unidades
- Actualización hormiga 5 La quinta hormiga encontró un camino de 57 unidades, que es un rastro mejor que el de los demás por lo que el depósito de feromonas será mayor que en los demás caminos ($\frac{1}{57} \approx 0.017543$)
- Actualización hormiga 6 Finalmente, el depósito que hará la sexta hormiga es de 0.0099009

  De este modo los rastros de feromonas actualizados quedan como se muestra en la siguiente tabla:

$\tau(1,2) = 0.123742$	$\tau(2,6) = 0.087523$
$\tau(1,3) = 0.097424$	$\tau(3,4) = 0.114990$
$\tau(1,4) = 0.122610$	$\tau(3,5) = 0.086196$
$\tau(1,5) = 0.0111$	$\tau(3,6) = 0.133266$
$\tau(1,6) = 0.097446$	$\tau(4,5) = 0.107525$
$\tau(2,3) = 0.087345$	$\tau(4,6) = 0.096097$
$\tau(2,4) = 0.00975$	$\tau(5,6) = 0.104889$
$\tau(2,5) = 0.142612$	

  Nótese que las aristas $(1,5)$ y $(2,4)$ se evaporaron ya que no fueron recorridos por ninguna hormiga. A modo de ejemplo detallaremos la actualización de la arista $(1,2)$:

$$\tau(1,2) = \tfrac{1}{10} * (1 - 0.22) + \tfrac{1}{99} + \tfrac{1}{122} + \tfrac{1}{57} + \tfrac{1}{101} \approx 0.123742$$

Paso 5. Hacemos $t = 1$ Como es la primera iteración debemos regresar a 2.

Segunda Iteración

Paso 3. Siguiendo el proceso de la misma manera que en la primera iteración encontraremos las siguientes soluciones.

1. Hormiga 1. Inicia en el nodo 1 y se construye la solución:

$$C_1 = \{(1,4),(4,5),(5,2),(2,6),(6,3),(3,1)\}$$

Con una distancia de 105 unidades

2. Hormiga 2. Inicia en el nodo 2 y se construye la solución:

$$C_2 = \{(2,5),(5,4),(4,6),(6,3),(3,1),(1,2)\}$$

Con una distancia de 101 unidades

3. Hormiga 3. Inicia en el nodo 3 y se construye la solución:

$$C_3 = \{(3,6),(6,4),(4,1),(1,2),(2,5),(5,3)\}$$

Con una distancia de 122 unidades

4. Hormiga 4. Inicia en el nodo 4 y se construye la solución:

$$C_4 = \{(4,1),(1,2),(2,5),(5,6),(6,3),(3,4)\}$$

Con una distancia de 57 unidades

5. Hormiga 5. Inicia en el nodo 5 y se construye la solución:

$$C_5 = \{(5,6),(6,3),(3,2),(2,1),(1,4),(4,5)\}$$

Con una distancia de 62 unidades

6. Hormiga 6. Inicia en el nodo 6 y se construye la solución:

$$C_6 = \{(6,3),(3,5),(5,2),(2,1),(1,4),(4,6)\}$$

Con una distancia de 122 unidades

Paso 4. El proceso de evaporación deja el siguiente rastro:

$$\begin{array}{ll}
\tau(1,2) = 0.156486 & \tau(2,6) = 0.077792 \\
\tau(1,3) = 0.095416 & \tau(3,4) = 0.107236 \\
\tau(1,4) = 0.155226 & \tau(3,5) = 0.083626 \\
\tau(1,5) = 0.008658 & \tau(3,6) = 0.173438 \\
\tau(1,6) = 0.076007 & \tau(4,5) = 0.119423 \\
\tau(2,3) = 0.084258 & \tau(4,6) = 0.101250 \\
\tau(2,4) = 0.007605 & \tau(5,6) = 0.115486 \\
\tau(2,5) = 0.164599 &
\end{array}$$

Paso 5. El valor de $t = 2$ de modo que represamos al paso 2.

Tercera Iteración

Paso 3. Siguiendo el proceso de la misma manera que en la primera iteración encontraremos las siguientes soluciones.

1. Hormiga 1. Inicia en el nodo 1 y se construye la solución:

$$C_1 = \{(1,4),(4,3),(3,6),(6,5),(5,2),(2,1)\}$$

Con una distancia de 57 unidades

2. Hormiga 2. Inicia en el nodo 2 y se construye la solución:

$$C_2 = \{(2,5),(5,6),(6,3),(3,4),(4,1),(1,2)\}$$

Con una distancia de 57 unidades

3. Hormiga 3. Inicia en el nodo 3 y se construye la solución:

$$C_3 = \{(3,6),(6,4),(4,1),(1,2),(2,5),(5,3)\}$$

Con una distancia de 122 unidades

4. Hormiga 4. Inicia en el nodo 4 y se construye la solución:

$$C_4 = \{(4,5),(5,2),(2,1),(1,6),(6,3),(3,4)\}$$

Con una distancia de 99 unidades

5. Hormiga 5. Inicia en el nodo 5 y se construye la solución:

$$C_5 = \{(5,2),(2,1),(1,4),(4,6),(6,3),(3,5)\}$$

Con una distancia de 122 unidades

6. Hormiga 6. Inicia en el nodo 6 y se construye la solución:

$$C_6 = \{(6,3),(3,2),(2,4),(4,1),(1,5),(5,6)\}$$

Con una distancia de 189 unidades

Paso 4. El proceso de evaporación deja el siguiente rastro:

$\tau(1,2) = 0.183641$	$\tau(2,6) = 0.0606780$
$\tau(1,3) = 0.074424$	$\tau(3,4) = 0.128833$
$\tau(1,4) = 0.177848$	$\tau(3,5) = 0.081622$
$\tau(1,5) = 0.052746$	$\tau(3,6) = 0.202155$
$\tau(1,6) = 0.069387$	$\tau(4,5) = 0.103251$
$\tau(2,3) = 0.071012$	$\tau(4,6) = 0.095368$
$\tau(2,4) = 0.052746$	$\tau(5,6) = 0.130458$
$\tau(2,5) = 0.189969$	

Paso 5. El valor de $t = 2$ de modo que regresamos al paso 2.

Cuarta Iteración

Paso 3. Siguiendo el proceso de la misma manera que en la primera iteración encontraremos las siguientes soluciones.

1. Hormiga 1. Inicia en el nodo 1 y se construye la solución:

$$C_1 = \{(1,4),(4,5),(5,2),(2,3),(3,6),(6,1)\}$$

   Con una distancia de 89 unidades

2. Hormiga 2. Inicia en el nodo 2 y se construye la solución:

$$C_2 = \{(2,5),(5,6),(6,3),(3,1),(1,4),(4,2)\}$$

   Con una distancia de 133 unidades

3. Hormiga 3. Inicia en el nodo 3 y se construye la solución:

$$C_3 = \{(3,6),(6,4),(4,5),(5,2),(2,1),(1,3)\}$$

   Con una distancia de 101 unidades

4. Hormiga 4. Inicia en el nodo 4 y se construye la solución:

$$C_4 = \{(4,1),(1,2),(2,5),(5,6),(6,3),(3,4)\}$$

   Con una distancia de 57 unidades

5. Hormiga 5. Inicia en el nodo 5 y se construye la solución:

$$C_5 = \{(5,2),(2,1),(1,4),(4,3),(3,6),(6,5)\}$$

   Con una distancia de 57 unidades

6. Hormiga 6. Inicia en el nodo 6 y se construye la solución:

$$C_6 = \{(6,3),(3,1),(1,4),(4,5),(5.2),(2,6)\}$$

   Con una distancia de 105 unidades

Paso 4. El proceso de evaporación deja el siguiente rastro:

$\tau(1,2) = 0.188229$	$\tau(2,6) = 0.056852$
$\tau(1,3) = 0.084994$	$\tau(3,4) = 0.135577$
$\tau(1,4) = 0.202088$	$\tau(3,5) = 0.063665$
$\tau(1,5) = 0.041142$	$\tau(3,6) = 0.230948$
$\tau(1,6) = 0.065358$	$\tau(4,5) = 0.111197$
$\tau(2,3) = 0.066625$	$\tau(4,6) = 0.084288$
$\tau(2,4) = 0.048660$	$\tau(5,6) = 0.144364$
$\tau(2,5) = 0.221443$	

Paso 5. El valor de $t = 2$ de modo que regresamos al paso 2.

Quinta Iteración

Paso 3. Siguiendo el proceso de la misma manera que en la primera iteración encontraremos las siguientes soluciones.

1. Hormiga 1. Inicia en el nodo 1 y se construye la solución:

$$C_1 = \{(1,2),(2,5),(5,6),(6,3),(3,4),(4,1)\}$$

   Con una distancia de 57 unidades

2. Hormiga 2. Inicia en el nodo 2 y se construye la solución:

$$C_2 = \{(2,5),(5,4),(4,1),(1,6),(6,3),(3,2)\}$$

   Con una distancia de 89 unidades

3. Hormiga 3. Inicia en el nodo 3 y se construye la solución:

$$C_3 = \{(3,4),(4,6),(6,5),(5,2),(2,1),(1,3)\}$$

   Con una distancia de 119 unidades

4. Hormiga 4. Inicia en el nodo 4 y se construye la solución:

$$C_4 = \{(4,5),(5,2),(2,1),(1,6),(6,3),(3,4)\}$$

   Con una distancia de 99 unidades

5. Hormiga 5. Inicia en el nodo 5 y se construye la solución:

$$C_5 = \{(5,2),(2,3),(3,1),(1,6),(6,4),(4,5)\}$$

   Con una distancia de 151 unidades

6. Hormiga 6. Inicia en el nodo 6 y se construye la solución:

$$C_6 = \{(6,3),(3,1),(1,4),(4,5),(5,2),(2,6)\}$$

   Con una distancia de 105 unidades

Paso 4. El proceso de evaporación deja el siguiente rastro:

$\tau(1,2) = 0.182867$	$\tau(2,6) = 0.053868$
$\tau(1,3) = 0.090845$	$\tau(3,4) = 0.141798$
$\tau(1,4) = 0.195932$	$\tau(3,5) = 0.049658$
$\tau(1,5) = 0.032009$	$\tau(3,6) = 0.228544$
$\tau(1,6) = 0.078938$	$\tau(4,5) = 0.124217$
$\tau(2,3) = 0.069826$	$\tau(4,6) = 0.080771$
$\tau(2,4) = 0.037954$	$\tau(5,6) = 0.138551$
$\tau(2,5) = 0.236156$	

Paso 5. El valor de $t = 2$ de modo que regresamos al paso 2.

Sexta Iteración

Paso 3. Siguiendo el proceso de la misma manera que en la primera iteración encontraremos las siguientes soluciones.

1. Hormiga 1. Inicia en el nodo 1 y se construye la solución:

$$C_1 = \{(1,4),(4,3),(3,6),(6,5),(5,2),(2,1)\}$$

   Con una distancia de 57 unidades

2. Hormiga 2. Inicia en el nodo 2 y se construye la solución:

$$C_2 = \{(2,5),(5,6),(6,3),(3,1),(1,4),(4,2)\}$$

   Con una distancia de 133 unidades

3. Hormiga 3. Inicia en el nodo 3 y se construye la solución:

$$C_3 = \{(3,6),(6,5),(5,2),(2,1),(1,4),(4,3)\}$$

   Con una distancia de 57 unidades

4. Hormiga 4. Inicia en el nodo 4 y se construye la solución:

$$C_4 = \{(4,1),(1,2),(2,5),(5,6),(6,3),(3,4)\}$$

   Con una distancia de 57 unidades

5. Hormiga 5. Inicia en el nodo 5 y se construye la solución:

$$C_5 = \{(5,2),(2,1),(1,4),(4.3),(3,6),(6,5)\}$$

   Con una distancia de 57 unidades

6. Hormiga 6. Inicia en el nodo 6 y se construye la solución:

$$C_6 = \{(6,3),(3,4),(4,1),(1,2),(2,5),(5,6)\}$$

   Con una distancia de 57 unidades

Paso 4. El proceso de evaporación deja el siguiente rastro:

$\tau(1,2) = 0.230355$	$\tau(2,6) = 0.042017$
$\tau(1,3) = 0.078378$	$\tau(3,4) = 0.198322$
$\tau(1,4) = 0.248065$	$\tau(3,5) = 0.038733$
$\tau(1,5) = 0.024967$	$\tau(3,6) = 0.273502$
$\tau(1,6) = 0.061571$	$\tau(4,5) = 0.096889$
$\tau(2,3) = 0.054464$	$\tau(4,6) = 0.063000$
$\tau(2,4) = 0.037124$	$\tau(5,6) = 0.203308$
$\tau(2,5) = 0.279440$	

Paso 5. El valor de $t = 6 = T^{max}$ por lo que el método termina.

La mejor solución encontrada en este punto fue con una distancia de 57 unidades $\{1,4,3,6,5,2,1\}$. Observemos que, en la última iteración, 5 de las 6 hormigas recorrieron la misma solución; es probable que con un número mayor de iteraciones todas las hormigas recorran el mismo camino, no obstante, también podemos notar que esta solución se encontró desde la segunda iteración. También podemos disminuir el número de hormigas, quizá no sería tan clara la convergencia a una solución específica pero podría ser encontrada.

2.2.2. Colonia de Hormigas para el problema del clique de cardinalidad máxima

Este problema también es posible representarlo de manera gráfica, sin embargo, no es tan clara la implementación del método, ya que aquí no buscamos una ruta. La parte que requiere nuestra atención es la construcción del clique.

Recordemos que un clique es un conjunto de nodos con la característica de que cualquier nodo de este conjunto es vecino de todos los demás. Por lo tanto, una vez que iniciamos con un nodo i de la gráfica, los siguientes elementos a conformar el clique se limitan a aquellos nodos que son vecinos de i (elementos que estarán en un conjunto C_{Cdtos}). La elección del siguiente nodo será la que decidamos de manera probabilista, una vez que se ha elegido un vértice j para añadirlo al clique se tiene que restringir la lista de candidatos a aquellos que son, además de vecinos de i vecinos de j. Este proceso se repetirá de manera consecutiva hasta que no queden opciones para añadir a los candidatos.

La probabilidad para elegir al candidato a formar parte del clique se calcula en [11] como:

$$P_k(p) = \frac{\tau(p)^\alpha}{\sum_{u \in C_{Cdtos}} \tau(u)^\alpha}$$

No considera información heurística, o lo que es lo mismo $\beta = 0$, no obstante, es posible usar información de este tipo pues el grado de un nodo con respecto a los elementos de C_{Cdtos} es algo que podemos considerar, si un nodo tiene más vecinos que otro es probable que se pueda construir un clique de mayor cardinalidad.

A diferencia del problema del agente viajero no podemos utilizar la función $\frac{1}{|C|}$ suponiendo que C en este caso corresponda al clique encontrado por la hormiga, esto debido a que entre más grande sea el valor de C nos debería resultar más atractivo y esa función no lo cumple. Podríamos aplicar, por ejemplo, la función: $\frac{1}{|X| + 1 - |C|}$

De esta manera, la construcción del clique se realizará de la siguiente manera:

Construcción del clique

1. Elegir un nodo $i \in X$ de manera aleatoria
2. Hacer $C_{Clique} = \{i\}$
3. Hacer $C_{Cdtos} = \{j : (i, j) \in A\}$
4. Elegir un nodo p de C_{Cdtos} con la siguiente probabilidad

$$P_k(p) = \frac{\tau(p)^\alpha \eta(p)^\beta}{\sum_{u \in C_{Cdtos}} \tau(u)^\alpha \eta(u)^\beta}$$

5. Hacer $C_{Clique} = C_{Clique} \cup \{p\}$ y $C_{Cdtos} = C_{Cdtos} \cap s : (p, s) \in A$

6. Verificar
   - Si $C_{Cdtos} = \emptyset$ Terminar, C_{Clique} es el clique construido
   - En caso contrario regresar a 4

Como mencionamos, en $\beta = 0$, pero con la intención de considerar el grado de los nodos se sugiere refinar un poco el paso 4. En lugar de elegir de manera probabilista un nodo del conjunto C_{Cdtos} se calcula primero el grado de los nodos de este conjunto y se elige al de mayor grado. Si sólo es uno se elige automáticamente, en caso de que sean dos o más con el mismo grado se aplica la elección probabilista.

Para nuestro ejemplo consideremos la gráfica que aparece en la figura 2.7:

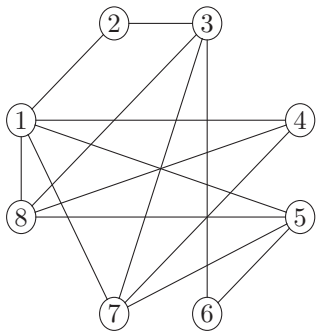

Figura 2.7: Ejemplo para el problema del clique de cardinalidad máxima

Supongamos que elegimos el nodo 3 para iniciar la construcción del clique. Los pasos a seguir para la construcción del clique serían los siguientes:

Paso 1. Iniciamos en el nodo 3

Paso 2. Hacer $C_{Clique} = \{3\}$

Paso 3. Sea $C_{Cdtos} = \{2, 6, 7, 8\}$

Paso 4. Calculamos el grado de cada uno con respecto al conjunto de candidatos. Pero como ninguno de ellos es adyacente entre sí todos tienen grado 0. De modo que elegimos de manera probabilista a alguno de ellos con la fórmula que aparece arriba. Al ser la primera iteración todos tienen la misma probabilidad de ser elegidos. Supongamos que que $\tau_0 = 0.12$ y $\alpha = 1$ de modo que la probabilidad de ser elegidos es $\frac{1}{4}$. Generamos un número aleatorio $r = 0.5081$ el cual nos indica que debemos elegir a 7 pues $r \in (0.5, 0.75]$

Paso 5. Actualizamos los conjuntos y tenemos que $C_{Clique} = \{1, 7\}$ y $C_{Cdtos} = \emptyset$.

Paso 6. Como no hay más candidatos de dónde elegir terminamos con un clique

de cardinalidad 2.

En esta iteración no se construyó un clique de cardinalidad máxima ya que los conjuntos $\{1, 4, 7\}$,$\{1, 4, 8\}$,$\{1, 5, 7\}$ y $\{1, 5, 8\}$ son cliques de cardinalidad 3. En el ejemplo del agente viajero sucedió que en la última iteración la mayoría de las hormigas eligieron la misma ruta, misma que resultó ser la que presentamos como mejor solución, sin embargo, en este caso la existencia de óptimos alternativos entorpecerá el método pues todos esos nodos podrían tener un rastro similar de feromonas y no se elegirá un clique en particular. Esto no significa que el método falle para este problema, pues se puede identificar al menos un clique de cardinalidad 3.

Usando el programa de sistema de hormigas para encontrar un clique usamos 3 escenarios. En el primero utilizamos 8 hormigas y 6 iteraciones, en los otros se usaron 4 hormigas y 10 y 6 iteraciones respectivamente, esto para tratar de encontrar algún ejemplo donde las hormigas converjan a una solución en la última iteración. La siguiente tabla muestra los resultados.

Hormigas	Iteraciones	Cliques
8	6	(1,7,4),(2,1),(3,6),(4,1,8),(5,1,7),(6,5),(7,1,4),(8,1,5)
4	10	(8,1,4),(6,3),(6,3),(3,8)
4	6	(4,1,7),(1,5,7)(4,1,7)(4,1,7)

El único ejemplo donde se encuentra un conjunto de hormigas que convergen a una solución es con 4 hormigas y 6 iteraciones, sin embargo, esto puede deberse más a una casualidad. Al ser 4 hormigas las distribuimos de manera aleatoria entre los nodos y en el caso de la última iteración sucede que 3 de las hormigas inician en el nodo 4, de donde sí eligen como nodos más atractivos a los otros nodos.

Capítulo 3

Métodos de Búsqueda Local

La estrategia principal de los métodos de búsqueda local consiste en construir una solución inicial y explorar sus alrededores en busca de otra solución con ciertas propiedades, en caso de encontrarla se procede a explorar sus alrededores, y así sucesivamente, hasta que se cumpla una condición que indique la detención del método.

La condición de paro varía según el método, al igual que las propiedades buscadas en cada exploración, lo que sí tienen en común es la idea de construir una solución inicial para comenzar, así como la idea de explorar los alrededores. El construir una solución inicial requiere la búsqueda de una manera de crear elementos pertenecientes a la región factible y preferentemente, que sean buenos para ahorrarle iteraciones al método; a pesar de ello pueden ser instrucciones de cualquier estilo. Consideremos el siguiente problema binario:

$$
\begin{aligned}
&Max\ z = 4x_1 + 2x_2 + 3x_3 + 6x_4 \\
&\text{s.a} \\
&6x_1 + 2x_2 + x_3 - 2x_4 \leq 5 \\
&4x_1 + 2x_3 + x_4 \leq 5 \\
&x_i \in \}
\end{aligned}
$$

Para determinar una solución inicial x_0 podríamos probar con las siguientes instrucciones que le darán valor de 1 a una de las variables:

1. Hacer $k = 1$
2. Hacer $x_k = 1$ y $x_j = 0\ \forall j \neq k$
3. Verificar que $(x_1, x_2, x_3, x_4) \in F_p$
   - Si es factible hacer $x_0 = (x_1, x_2, x_3, x_4)$
   - En caso contrario hacer $k = k + 1$ e ir a 2

Siguiendo estas instrucciones la solución inicial que aplicaríamos sería $x_0 = (0, 1, 0, 0)$; una idea más elaborada consiste en hacer lo siguiente:

1. Hacer $x_1 = x_2 = x_3 = x_4 = 0$ y $k = 1$
2. Hacer $x_k = 1$ y verificar que $(x_1, x_2, x_3, x_4) \in F_p$
   - Si $(x_1, x_2, x_3, x_4) \in F_p$ conservar $x_k = 1$
   - Si $(x_1, x_2, x_3, x_4) \notin F_p$ hacer $x_k = 0$
3. Hacer $k = k + 1$ e ir a 2

Cuyo resultado será la solución $(0, 1, 0, 1)$ que tiene un valor de 8 en su función objetivo, mejor que el 4 obtenido por el primer método. Si consideramos ordenar las variables según su aportación a la función objetivo y repitiendo el segundo proceso se podría obtener un mejor resultado ($x_0 = (1, 0, 0, 1)$, $z(x_0) = 10$), más aún, si hiciéramos uso de un método glotón (que es prácticamente lo que hicimos para la tercera solución) para obtener x_0 sería más probable encontrar una buena solución inicial sin haber hecho un gasto excesivo de recursos pues sabemos que este método no requiere un esfuerzo muy grande. De hecho, la idea de combinar un heurístico constructivo con uno de búsqueda local suele ser la más aplicada, esto porque se obtiene una buena solución inicial. El uso de un método distinto a un heurístico glotón valdría la pena en casos donde éste proporcione malas soluciones.

Una vez que tenemos la solución inicial a partir de la cual exploraremos sus alrededores, es momento de definir a qué nos referimos por alrededores. En cálculo definimos una vecindad para un número real x como el conjunto $N'_\epsilon(x) = \{x' : |x' - x| < \epsilon\}$ para algún $\epsilon > 0$, que denota al conjunto de números que se encuentran a cierta distancia de x. El concepto general de vecindad conserva esa idea de incluir sólo a los elementos de los "alrededores", pero serán aquellos a los que podamos acceder por medio de un mecanismo de intercambio, en el ejemplo del número x el mecanismo consiste en sumar o restar un número $a < \epsilon$.

No obstante, este concepto de vecindad tiene un problema; para la implementación de los métodos los vecinos deben ser factibles, lo cual no necesariamente sucederá con la definición de anterior. Por lo tanto, ajustaremos la definición y diremos que la vecindad $N(x)$ será formada por el conjunto dado por una función que se limite a considerar soluciones factibles, que será del siguiente estilo:

$$N : F_p \to 2^{F_p}$$

Es decir, dado un mecanismo de intercambio excluiremos de la vecindad de manera automática a aquellos elementos que estén fuera de F_p. Estos mecanismos deberán personalizarse según el tipo de problema que se presente, por ejemplo, si las soluciones del problema pueden ser representadas como un vector en $\mathbb{R}^n$ con valores enteros, podemos definir una vecindad así:

$$N_m(x) = \{x' | x' = (x_1, x_2, ..., x_{j+k}, ..., x_n), \text{para } j = 1, 2, ..., n \text{ y para } k = 1, 2, 3.., m\}$$

Por ejemplo: Para la solución $x = (1, 2)$ tenemos la vecindad:
$N_1(x) = \{(0, 2), (2, 2), (1, 1), (1, 3)\}$

Por otro lado, si el problema es representado por una gráfica, como es el caso del problema del agente viajero, un vecino puede ser formado por el intercambio de k aristas en el ciclo. En la figura 3.1 se muestra un ciclo hamiltoniano y un vecino producido por un intercambio de 2 aristas.

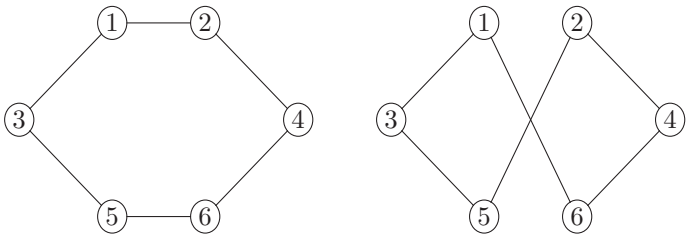

Figura 3.1: Ejemplo para definir un vecino en una gráfica

Al implementar el método, la vecindad adquiere importancia principalmente por dos aspectos: el tamaño y la exploración. Una vecindad grande obtendrá un mejor resultado, pero tiene inconvenientes; pongamos un ejemplo, si tenemos una solución x y su vecindad $N(x)$ corresponde al resto de F_p significa que podremos encontrar el óptimo a través de x, sin embargo, el precio será la exploración de toda la región, que en principio es lo que queremos evitar. Por lo tanto, una vecindad no debe ser muy grande así como tampoco debe ser muy pequeña, pues entorpecerá el método al incrementar la posibilidad de caer en un óptimo local. Lo ideal sería un punto intermedio, es decir, una vecindad que no sea muy pequeña, pero que se pueda explorar en un tiempo razonable. En el problema del agente viajero un intercambio de 2 o 3 aristas es suficiente para una exploración buena, mientras que 4 o más producen una mejora mínima comparada con el incremento de tiempo en la ejecución del método.

El otro aspecto, la exploración, también influye en el tiempo de resolución. La primera manera de explorar una vecindad es analizando cada elemento de la misma, y elegir el mejor candidato. En vecindades pequeñas esto ayudaría a realizar un mejor análisis sin sacrificar el tiempo; la ventaja de esto es que la vecindad es explorada con una mejor calidad. Otra alternativa es realizar exploraciones parciales, por ejemplo, elegir elementos al azar hasta encontrar alguno con las características deseadas; este tipo de exploración ayuda en la reducción del tiempo de ejecución, en especial para vecindades grandes, la desventaja principal es que la calidad de la solución disminuye.

Algunos otros elementos comunes a los métodos de búsqueda local están relacionados con el gasto de recursos. Al hacer la comparación entre la solución actual y sus vecinos, el valor de la función objetivo de muchas soluciones puede haber sido obtenido con anterioridad; si estos valores se almacenaran en listas

para ser revisadas en iteraciones posteriores tal gasto se verá reducido.

El uso de listas no es únicamente válido para no realizar el cálculo de la función objetivo de soluciones. El método de recocido simulado utiliza la generación de valores de una función exponencial, y en ese caso el uso de listas también es aplicable.

En las secciones posteriores hablaremos sobre los elementos que definen a cada heurístico de búsqueda local, y antes de describir los métodos para un problema en particular implementaremos los métodos en el problema de maximizar la siguiente función:

x	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$f(x)$	227	56	149	113	7	203	28	83	111	40	105	20	70	90	60

Partiremos de una solución $x_0 = 7$ que es de las más bajas y cercana a $x = 6$ cuyo valor es parecido al óptimo, lo cual esperamos que atrape a algún método mostrándonos una de sus fallas: quedar atrapados en un óptimo local.

3.1. Mejoras Sucesivas

Como ya hemos visto, la exploración de la vecindad es la característica que define a los métodos de búsqueda local. El método de Mejoras Sucesivas la realiza por medio de un concepto bastante conocido: el ensayo y el error. Una cualidad que posee el óptimo x^* de cualquier problema es que no hay solución mejor y, en consecuencia, no hay nadie mejor en la vecindad $N(x^*)$; de modo que, si tenemos una solución x tal que algún vecino $\tilde{x}$ es mejor se hace evidente la afirmación de que x no es una solución buena, pero si x es mejor que cualquier $\tilde{x} \in N(x)$, por el contrario x será un buen candidato a ser el óptimo. Aunque no necesariamente será el óptimo, el método falla por la deducción que hicimos antes:

Si x^* es mejor que cualquier elemento de la región factible $\Longrightarrow$ x^* es mejor que cualquier $x \in N(x^*)$

Como mencionamos antes, la calidad de estos métodos se ve disminuida cuando exploran vecindades de óptimos locales; ya que si un elemento es mejor que cualquiera de los vecinos no necesariamente será el óptimo global, en gran parte depende del tamaño de la vecindad. Pese a ser una falla a tomar en cuenta, la estrategia es buena, tan buena que el algoritmo simplex emplea la misma estrategia y aprovechándose de más elementos sí logra encontrar el óptimo.

A grandes rasgos el método de Mejoras Sucesivas se puede implementar definiendo los siguientes pasos:

Método de Mejoras Sucesivas.

1. Obtener una solución inicial x_0 (Puede incluso utilizarse un método constructivo)
2. Sea $x_0 = x_{actual}$
3. Explorar la vecindad $N(x_{actual})$
   - Si existe un x tal que $z(x)$ mejor que $z(x_{actual})$ Hacer $x_{actual} = x$ y regresar a 3
   - En otro caso hacer $x_{actual} = x_{final}$

Este método tiene un criterio de paro muy intuitivo, al no encontrar algún elemento mejor en la vecindad de una solución considera a ésta como la mejor.

Para ejemplificar el funcionamiento del método de Mejoras Sucesivas aplicaremos dos casos al ejemplo descrito al inicio del capítulo, el primero cuando se realiza una exploración completa y el segundo por medio de una exploración aleatoria.

Los resultados del primer proceso se muestran en la siguiente tabla:

x	$f(x)$	$N(x)$	Vecino Mejor
7	28	5,6,8,9	6, $f(6) = 203$
6	203	4,5,7,8	—

Podemos observar que desde la primera iteración el método toma como solución actual un óptimo local, que le impide explorar más vecindades debido que con $x = 6$ no fue posible encontrar algún vecino mejor, sin embargo resultó ser la segunda mejor solución, lo cual es bastante cercano al óptimo.

El resultado del segundo caso[1] es el que aparece en la siguiente tabla:

x	$f(x)$	Núm. Aleatorio	Vecindad	Vecino examinado
7	28	0.792	5,6,8,9	8
8	83	0.145	6,7,9,10	6
6	203	0.399	4,5,7,8	5
6	203	0.137	4,7,8	4
6	203	0.914	7,8	8
6	203	0.408	7	7

Al revisar menos elementos de la vecindad es esperarse que se obtuviera la solución con un menor número de operaciones; sin embargo, observemos que incluso pudo haberse escapado del óptimo local con otra selección de números aleatorios, eligiendo vecinos no tan mejores y alejándose un poco más del óptimo; de cualquier modo queda en evidencia que éste método puede corre el riesgo de encontrar un óptimo local. Con el objetivo de escapar de ellos, el método de

[1]Recordemos que éste consiste en calcular la función de uno de los vecinos de manera aleatoria, por lo que generamos un número aleatorio para elegir a cada vecino. En el primer caso elegimos a 5 si el número generado estuviera entre [0,0.25), a 6 si estuviera entre [0.25,0.5), a 8 si estuviera entre [0.5,0.75) y finalmente a 9 si el número fuera mayor a 0.75. En la tabla se muestra que el número obtenido queda incluido en este último caso por lo que elegimos a 9

Mejoras Sucesivas sufrió algunos cambios que dieron lugar a otros heurísticos como Búsqueda Tabú y Recocido Simulado.

A modo de ejemplo resolveremos algunos problemas por medio de este método, empezando por el problema del agente viajero.

3.1.1. Mejoras sucesivas para el problema del agente viajero

Empezaremos con este problema porque es el que ya tenemos en su mayor parte resuelto. Al inicio de este capítulo mencionamos que la vecindad está definida por un $k-$intercambio, en particular nosotros usaremos $k = 2$. La exploración de la vecindad la haremos de 2 maneras, primero mediante una exploración completa eligiendo al vecino mejor y posteriormente una exploración aleatoria, avanzando cuando el vecino sea mejor que la solución actual.

El intercambio requiere que pongamos un poco de atención, por ejemplo, dijimos que éste se realiza quitando 2 aristas de la solución actual y reemplazándolos con otros, como en el caso de la gráfica que se muestra en la figura 3.2:

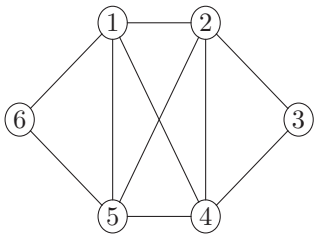

Figura 3.2: Ejemplo para el problema del agente viajero

Si el ciclo está dado por el conjunto de aristas (1,2)(2,3)(3,4)(4,5)(5,6)(6,1) podemos retirar las aristas (1,2) y (5,4) y sustituirlas por (1,4) y (2,5), por otro lado las aristas (1,5) y (2,4) no pueden ser porque se crean dos sub gráficas. Por lo tanto, al reemplazar las aristas (i, j) y (m, n) sólo una de las combinaciones (i, n) y (j, m) ó 4(i, m) y (j, n) será factible.

También debemos observar que este intercambio sólo se puede dar entre aristas que no sean adyacentes, usando el ejemplo anterior tomemos a (1,2) y (2,3). Las posibles parejas del intercambio serían (1,2) y (2,3) ó (1,3) y (2,2), ninguna de las cuales tiene sentido.

Por último, no está de más recordar que un vecino puede estar definido por un intercambio y no ser factible, de modo que también debemos revisar la factibilidad.

La representación gráfica del ciclo no es necesaria, podemos sustituirla por alguna notación como el conjunto de aristas que lo forman. Por ejemplo:

$$(a_{12}, a_{23}, a_{34}, a_{45}, a_{56}, a_{61})$$

Representa el mismo ciclo al que nos hemos referido.

Una de las desventajas de los métodos de búsqueda local es que durante la exploración se puede la misma solución más de una vez; si s_1 es solución actual y tiene una solución vecina s' calculamos el valor de su función objetivo, y si posteriormente revisamos una solución s_2 que también tiene como vecino a s' volverá a ser revisada. Para corregir esto puede usarse una lista donde se almacenen estos resultados o podríamos buscar una forma más eficiente de calcular la función objetivo (en este caso la "distancia"del ciclo).

Sabemos que en este problema calculamos la distancia[2] total del recorrido dado por la solución s que se obtiene con la fórmula siguiente:

$$d(s) = \sum_{a_{ij} \in s} c_{ij}$$

Adicionalmente sabemos que se hace un intercambio de 2 aristas, supongamos que son las a_1 y a_2. Y para completar el ciclo añadimos a $\tilde{a}_1$ y $\tilde{a}_2$, de manera que la distancia de la nueva solución vecina se puede obtener como:

$$\sum_{a_{ij} \in s} c_{ij} - c_{a_1} - c_{a_2} + c_{\tilde{a}_1} + c_{\tilde{a}_2}$$

Que es lo mismo que:

$$d(s) - c_{a_1} - c_{a_2} + c_{\tilde{a}_1} + c_{\tilde{a}_2}$$

De hecho, esto nos permite tener una manera rápida de verificar si este vecino es mejor o peor que s. Si $-c_{a_1} - c_{a_2} + c_{\tilde{a}_1} + c_{\tilde{a}_2}$ es mayor que cero implica que la distancia de las aristas intercambiadas es mayor y en consecuencia se obtiene una solución peor. Por el contrario si esta suma es negativa este vecino es mejor. Para ejemplificar le daremos distancias a las aristas de la gráfica anterior (figura 3.2) en la figura 3.3:

El ciclo que definimos se recorre en una distancia de 326 unidades. Si calculamos la diferencia entre la distancia de las aristas agregadas y las que quitamos tenemos: $8 + 29 - 24 - 91 = -78$ Lo que quiere decir que es mejor pues la distancia de esta nueva solución es $326 - 78 = 248$.

Si realizamos una exploración completa de la vecindad entonces el método de mejoras sucesivas para el problema del agente viajero es el siguiente:

[2]EL valor c_{ij} asociado a la arista (i,j) suele referirse como distancia aunque bien puede representar un costo o algún otro valor

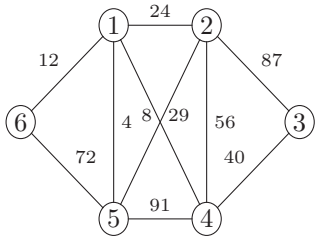

Figura 3.3: Gráfica con distancia entre las aristas

Método de Mejoras Sucesivas para el Agente Viajero.

1. Obtener una solución inicial s_0. Hacer $s = s_0$
2. Encontrar la vecindad $N_2(s)$ y calcular el cambio de distancia entre s y sus vecinos
3. Hacer $N_2^*(s) = \{s' | s' \in N_2(s) \text{ y } d(s') \leq d(s)\}$
4. Calcular la cardinalidad de $N_2^*(s)$
   - Si $N_2^*(s) = \emptyset$ terminar. s es la solución encontrada
   - En otro caso elegir un $s' \in N_2^*(s)$, hacer $s = s'$ y volver a 2

Si no realizamos una exploración completa de la vecindad el método constará de 5 pasos que describimos a continuación:

Segundo Método de Mejoras Sucesivas para el Agente Viajero.

1. Construir una solución inicial s_0. Hacer $s = s_0$
2. Obtener los vecinos de s, sea $N_2(s)$ ese conjunto
3. Elegir de manera aleatoria un $s' \in N_2(s)$
4. Calcular el cambio de distancia entre s y s'
   - Si s' es mejor hacer $s = s'$ y volver a 2
   - En otro caso hacer $N_2(s) = N_2(s) \setminus \{s'\}$ y pasar a 5
5. Si $N_2(s) = \emptyset$ terminar, la solución encontrada es s, en caso contrario regresar a 3

La solución inicial como hemos mencionado puede construirse con un método del capítulo anterior, en particular el Glotón por su rapidez, aunque cualquier solución factible servirá.

Probaremos el primer método con el siguiente ejemplo:
Cuyas distancias de las aristas son las siguientes:

$$\begin{array}{lll} (1,2) = 10 & (2,3) = 10 & (3,5) = 80 \\ (1,3) = 21 & (2,4) = 16 & (3,6) = 15 \\ (1,4) = 34 & (2,5) = 19 & (4,5) = 10 \\ (1,5) = 14 & (2,6) = 22 & (4,6) = 94 \\ (1,6) = 150 & (3,4) = 10 & (5,6) = 10 \end{array}$$

Este ejemplo se parece a uno que mencionamos en el heurístico Glotón, hay aristas que tienen un peso muy bajo y la última arista que el glotón mete

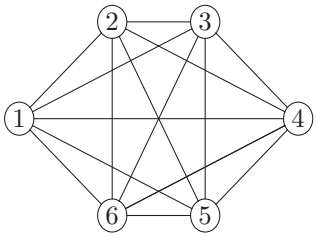

Figura 3.4: Ejemplo para resolver el problema del agente viajero usando el método de mejoras sucesivas

para completar el ciclo tiene un peso muy grande. Si bien no nos da la peor solución posible ($s_0 = \{(1,2)(2,3)(3,4)(4,5)(5,6)(6,1)\}$ con una distancia de 200 unidades), cuya gráfica se puede apreciar en la figura 3.5, nos da una solución mala que podemos mejorar.

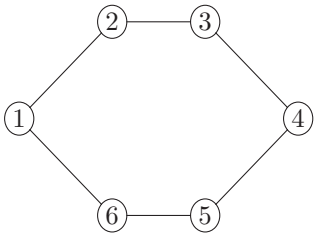

Figura 3.5: Representación gráfica de s_0

Es necesario saber cuántos vecinos tiene cada solución; sabemos que de las 6 aristas que forman el ciclo hamiltoniano debemos quitar a 2 y añadir otras dos más así que a lo más hay $C_2^6 = 15$ opciones, pero debemos descartar aquellas que toman dos aristas adyacentes pues no nos darían una solución vecina (por ejemplo, descartamos las aristas $(5,6)$ y $(4,5)$ ya que al hacer el intercambio tendremos como "vecino.ª s_0) que son 6, de modo que cada solución tendrá un total de 9 vecinos. Recordemos que al principio del capítulo se habló de un k-intercambio como mecanismo de vecindad para este problema, y como especificamos en la descripción del método de mejoras sucesivas para el problema del agente viajero consideraremos $k = 2$.

Ahora describiremos en su totalidad la aplicación del método:

Primera Iteración

Paso 1. Obtener una solución inicial. Como es posible construir una con un glotón nos quedamos con la solución $s' = \{(1,2)(2,3)(3,4)(4,5)(5,6)(6,1)\}$

Paso 2. Obtener los vecinos. De modo que $N_2(s) = \{$ $\{(1,3),(3,2),(2,4),(4,5),(5,6),(6,1)\}$, $\{(1,2),(2,4),(4,3),(3,5),(5,6),(6,1)\}$, $\{(1,2),(2,3),(3,5),(5,4),(4,6),(6,1)\}$, $\{(1,2),(2,3),(3,4),$

$(4,6),(6,5),(5,1)\}$, $\{(1,5),(5,6),(6,4),(4,3),(3,2),(2,1)\}$, $\{(1,2),(2,6),(6,5),(5,4),(4,3),(3,1)\}$, $\{(1,4),(4,3),(3,2),(2,5),(5,6),(6,1)\}$, $\{(1,2),(2,5),(5,4),(4,3),(3,2),(2,1)\}$, $\{(1,2),(2,3),(3,6),(6,5),(5,4),(4,1)\}$ } Cuya representación gráfica aparece en la figura 3.6.

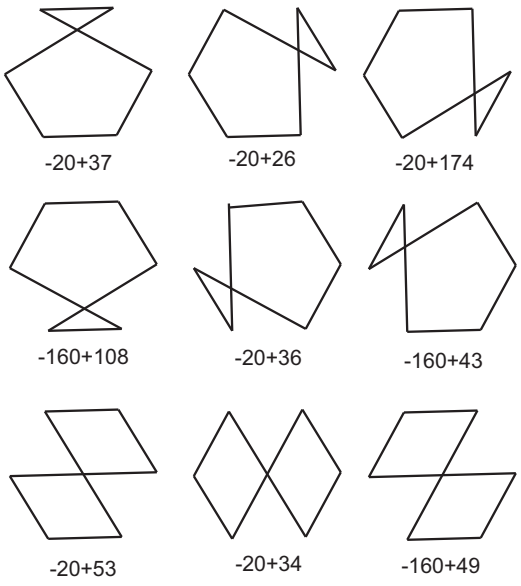

Figura 3.6: vecinos de s

Paso 3. Observemos que los únicos vecinos que presentan una mejora son los que quitan al arco $(1,6)$,es decir, el que el método glotón metió al final. $N_2^*(s) = \{$ $\{(1,2),(2,3),(3,4),(4,6),(6,5),(5,1)\}$, $\{(1,2),(2,6),(6,5),(5,4),(4,3),(3,1)\}$, $\{(1,2),(2,3),(3,6),(6,5),(5,4),(4,1)\}$ }

Paso 4. Como el conjunto es distinto del vació debemos elegir una solución mejor. Sólo para tener un criterio específico elegiremos al mejor vecino de todos, que en este caso es $s' = \{(1,2),(2,6),(6,5),(5,4),(4,3),(3,1)\}$ con un valor de $z(s') = 200 - 117 = 83$.

Ahora hacemos $s = s'$ y regresamos a 2.

Segunda Iteración

Paso 2. La nueva vecindad es la siguiente: $N_2(s) = \{$ $\{(1,3),(3,4),(4,5),(5,2),(2,6),(6,1)\}$, $\{(1,3),(3,4),(4,2),(2,6),(6,5),(5,1)\}$, $\{(1,3),(3,2),(2,6),(6,5),(5,4),(4,1)\}$, $\{(1,3),(3,4),(4,6),(6,5),(5,2),(2,1)\}$, $\{(1,3),(3,6),(6,5),(5,4),(4,2),(2,1)\}$, $\{(1,2),(2,3),(3,4),(4,5),(5,6),(6,1)\}$, $\{(1,2),(2,6),(6,4),(4,5),(5,3),(3,1)\}$, $\{(1,2),(2,6),(6,3),(3,4),(4,5),(5,1)\}$, $\{(1,2),(2,6),(6,5),(5,3),(3,4),(4,1)\}$ }

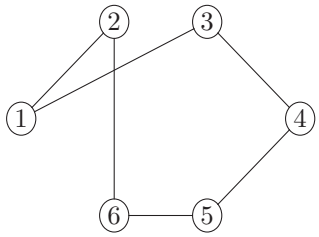

Figura 3.7: Solución elegida: s'

Paso 3. En este caso sólo existe un único ciclo mejor que s, sin embargo, $N_2^*(s) \neq \emptyset$

Paso 4. Hacemos $s = \{(1,2),(2,6),(6,3),(3,4),(4,5),(5,1)\}$ que es la única solución mejor y cuya distancia es $z(s) = 83 - 2 = 81$ y pasamos a 2

Tercera Iteración

Paso 2. La nueva vecindad, así como la diferencia con respecto a s se muestran en la siguiente tabla:

$\{(1,6),(6,2),(2,3),(3,4),(4,5),(5,1)\}$	$-10-15+150+10$
$\{(1,3),(3,6),(6,2),(2,4),(4,5),(5,1)\}$	$-10-10+21+16$
$\{(1,4),(4,3),(3,6),(6,2),(2,5),(5,1)\}$	$-10-10+34+19$
$\{(1,2),(2,3),(3,6),(6,4),(4,5),(5,1)\}$	$-22-10+10+94$
$\{(1,2),(2,4),(4,3),(3,6),(6,5),(5,1)\}$	$-22-10+16+10$
$\{(1,2),(2,5),(5,4),(4,3),(3,6),(6,1)\}$	$-22-14+150+19$
$\{(1,2),(2,6),(6,4),(4,3),(3,5),(5,1)\}$	$-15-14+21+10$
$\{(1,2),(2,6),(6,5),(5,4),(4,3),(3,1)\}$	$-15-14+21+10$
$\{(1,2),(2,6),(6,3),(3,5),(5,4),(4,1)\}$	$-10-14+34+80$

Paso 3. Y nuevamente sólo existe un elemento en $N_2^*(s)$ que es el ciclo $\{(1,2),(2,4),(4,3),(3,6),(6,5),(5,1)\}$

Paso 4. Elegimos esta solución como la siguiente a donde nos moveremos y regresamos a 2. La distancia en este momento es de $81 - 6 = 75$

Cuarta Iteración

Paso 2. La solución $s = \{(1,2),(2,4),(4,3),(3,6),(6,5),(5,1)\}$ tiene la siguiente vecindad con sus respectivos cambios con respecto a la distancia actual:

$$\begin{array}{ll}
\{(1,4),(4,2),(2,3),(3,6),(6,5),(5,1)\} & -10-10+10+34 \\
\{(1,3),(3,4),(4,2),(2,6),(6,5),(5,1)\} & -10-15+21+22 \\
\{(1,6),(6,3),(3,4),(4,2),(2,5),(5,1)\} & -10-10+150+19 \\
\{(1,5),(5,6),(6,4),(4,3),(3,2),(2,1)\} & -16-15+10+94 \\
\{(1,2),(2,6),(6,3),(3,4),(4,5),(5,1)\} & -16-10+22+10 \\
\{(1,2),(2,5),(5,6),(6,3),(3,4),(4,1)\} & -16-14+34+19 \\
\{(1,2),(2,4),(4,6),(6,3),(3,5),(5,1)\} & -10-10+80+94 \\
\{(1,2),(2,4),(4,5),(5,6),(6,3),(3,1)\} & -10-14+10+21 \\
\{(1,2),(2,4),(4,3),(3,5),(5,6),(6,1)\} & -15-14+150+80
\end{array}$$

Paso 3. En esta iteración no tenemos ninguna solución cuya distancia sea menor por lo que el conjunto $N_2^*(s)$ es vacío

Paso 4. Al no haber vecinos mejores a la solución actual hemos terminado con la ruta $\{(1,2),(2,4),(4,3),(3,6),(6,5),(5,1)\}$ que se recorre una distancia de 75 unidades. Misma que podemos verificar sumando la distancia de cada arco: 10+16+10+15+10+14.

La solución se encontró después de 4 iteraciones con una diferencia muy superior a la distancia de la solución inicial. Revisamos un total de 36 soluciones, de las cuales 29 eran distintas, es decir, casi un 20 % de las soluciones examinadas (7 soluciones) fueron evaluadas dos veces, de modo que el comparar únicamente la diferencia representa una ventaja importante, o bien utilizar listas sería más eficiente. También observemos que en este problema elegimos como solución siguiente a la mejor de todas las soluciones que presentaban una disminución en la función objetivo, aunque en realidad sólo usamos ese criterio en la primera iteración pues en las demás sólo había una solución.

3.1.2. Mejoras Sucesivas para el problema de Secuencia de Tareas

El problema de Secuencia de Tareas o Asignación de Tareas consiste en encontrar una orden para realizar un conjunto de actividades, cada una debe hacerse antes de un tiempo determinado o se incurre en un gasto por penalización. El objetivo es determinar la asignación que incurra en el gasto mínimo.

Este problema lo mencionamos cuando demostramos que pertenece a la clase de los problemas NP Duros, usaremos la misma notación que en ese capítulo, así que diremos que el número de tareas que debemos ordenar es t, que tienen un tiempo de realización τ_i $(i = 1, ..., t)$, una penalización P_i y una fecha límite para realizarse d_i. Un ejemplo es el que se muestra en la siguiente tabla:

Tarea	Tiempo de realización (días)	Fecha límite (día)	Penalización (por día)
1	10	24	14
2	28	58	16
3	9	21	9
4	15	22	10
5	25	32	20

Donde la actividad 1 debe hacerse antes del día 24 y tardamos 10 días en realizarla, de modo que si nos pasamos de ese día tendremos una penalización de 14 unidades por día de retraso. Taha en su libro [12] maneja un valor llamado costo de retención con el que se refiere a un costo que se tiene por terminar una tarea antes de tiempo, que es un caso más general pues en nuestro ejemplo el costo de retención es igual a cero. Sin embargo, el método que aplicaremos se puede utilizar para ambos casos.

Lo primero que necesitamos es encontrar una vecindad para este tipo de problemas, si tenemos la lista de actividades representada como un vector (por ejemplo (1,2,3,4,5) que indica que la actividad 1 se hace en primer lugar, seguida de la 2, etc.) hacer un intercambio de actividades es una opción, por ejemplo intercambiar dos actividades adyacentes ((2,1,3,4) sería un vecino de la solución anterior).

Ahora analicemos cómo se obtendría el gasto por penalización de un vecino. La secuencia (1,2,3,4,5) genera un gasto de $1,734$ unidades, ya que a partir de la tercera actividad tenemos retrasos: la tercera se realiza 26 días después, la cuarta después de 41 y la quinta actividad se realiza al día 87 que es 55 después del requerido. Sabiendo ese resultado notemos que el vecino (2,1,3,4,5) generará el mismo gasto por las actividades 3,4 y 5, ya que esas se terminarán en la misma fecha, sólo necesitamos calcular el gasto generado o ahorrado con el intercambio. Dado que la actividad 2 se realiza primero, la terminamos mucho antes del día 58, sin embargo, al realizar después la actividad 1 no se puede terminar a tiempo, es decir, realizar primero la actividad 2 nos ocasiona un retraso de 14 días en la actividad 1, de modo que la solución vecina debe ser 196 unidades más costosa que la primera.

Si tomamos como vecinos a aquellas soluciones que se produzcan del intercambio de cualesquiera dos tareas el cálculo del gasto no sería tan fácil porque se modifican más fechas, en ese caso valdría la pena utilizar listas, motivo por el cual consideraremos como vecinas a las soluciones que se generen de un intercambio de tareas adyacentes. Es decir, hacer intercambios como el que acabamos de describir pasando de la secuencia (1,2,3,4,5) a la (2,1,3,4,5) donde se intercambiaron a las tareas 1 y 2 que se pretenden realizar una después de la otra.

Las instrucciones para implementar el método de Mejoras Sucesivas en este problema son las siguientes:

Método de Mejoras Sucesivas para el problema de Asignación de Tareas.

1. Construir una solución inicial s_0 hacer $s = s_0$
2. Encontrar la vecindad de s y calcular el cambio en el gasto de cada vecino
3. Si todos los vecinos de s presentan un cambio mayor o igual que cero en el cambio terminar, s es la mejor solución encontrada. En otro caso ir a 4
4. Elegimos algún $s' \in N(s)$ cuyo cambio sea menor que cero
5. Hacer $s = s'$ y regresar a 2

Resolveremos el ejemplo de arriba con la solución inicial $s_0 = (1, 2, 3, 4, 5)$ y $z(s_0) = 1734$

Primera Iteración

Paso 1. Sea $s_0 = (1, 2, 3, 4, 5)$ y hacemos $s = s_0$

Paso 2. Los vecinos de s son 4. $N(s) = \{(2, 1, 3, 4, 5), (1, 3, 2, 4, 5), (1, 2, 4, 3, 5), (1, 2, 3, 5, 4)\}$. El primer vecino es resultado de intercambiar la tarea 1 y 2 que como mencionamos arriba producen una diferencia de 196 unidades, es decir; el segundo vecino intercambia las actividades 2 y 3, hacer primero la actividad 3 ocasionará que se termine en el día 19 con lo que elimina la penalización y la actividad 2 se termina en el día 47 así que tampoco genera un costo de modo que esta solución sí representa una disminución en la penalización de 234 unidades (que son las que se incurren cuando la tarea 3 se hace en tercer lugar). De manera análoga podemos verificar que el tercer vecino incrementa el costo en 45 unidades y el cuarto lo disminuye en 50

Paso 3. Como no todos los vecinos tienen un aumento en el costo pasamos a 4

Paso 4. Elegimos al vecino $(1, 3, 2, 4, 5)$ porque presenta la mayor disminución en el gasto, por lo que ahora este será el valor de s. Y $z(s) = 1734 - 234 = 1500$

Paso 5. $s = (1, 3, 2, 4, 5)$

Segunda Iteración

Paso 2. El conjunto de vecinos ahora es $N(s) = \{(3, 1, 2, 4, 5), (1, 2, 3, 4, 5), (1, 3, 4, 2, 5), (1, 3, 2, 5, 4)\}$ cuyos cambios con respecto a s son 0,234,-216,-50. Notemos que el segundo vecino es s_0 de modo que ya podíamos saber que era una solución peor.

Paso 3. Los últimos vecinos sí presentan una mejora de modo que pasamos a 4

Paso 4. Las soluciones (1,3,4,2,5) y (1,3,2,5,4) son mejores, elegiremos a la primera porque es la que presenta una mayor mejora.

Paso 5. Hacemos $s = (1, 3, 4, 2, 5)$

Tercera Iteración

Paso 2. La nueva vecindad es $N(s) = \{(3, 1, 4, 2, 5), (1, 4, 3, 2, 5), (1, 3, 2, 4, 5), (1, 3, 4, 5, 2)\}$ con cambios de 0,27,216 y -160 respectivamente.

Paso 3. Aún queda un vecino mejor $(1, 3, 4, 5, 2)$

Paso 4. Elegimos al único vecino que presenta una mejoría

Paso 5. Hacemos $s = (1, 3, 4, 5, 2)$

Cuarta Iteración

Paso 2. Los vecinos de $(1, 3, 4, 5, 2)$ son: $(3, 1, 4, 5, 2)$ cuyo costo no varía,$(1, 4, 3, 5, 2)$ que aumenta en 27 unidades el costo,$(1, 3, 5, 4, 2)$ que disminuye en 50 y $(1, 3, 4, 2, 5)$ que es el vecino anterior y por lo tanto el aumento es de 160.

Paso 3. Como aún hay un vecino mejor pasamos al paso 4

Paso 4. Hacemos $s' = (1, 3, 5, 4, 2)$ pues es la única solución mejor

Paso 5. Sea $s = s'$ y volvemos a 2

Quinta Iteración

Paso 2. La vecindad en esta iteración es $\{(3, 1, 5, 4, 2), (1, 5, 3, 4, 2), (1, 3, 4, 5, 2), (1, 3, 5, 2, 4)\}$ cuyos respectivos cambios en el costo son de 0, 27, 50 y 40.

Paso 3. Como todos los vecinos son soluciones peores o iguales a la actual hemos terminado. La solución $s = (1, 3, 5, 4, 2)$ es la mejor encontrada.

$z(s) = 1734 - 234 - 216 - 160 - 50 = 1074$

En este ejemplo es posible notar otro riesgo al usar el método de Mejoras Sucesivas: caer en un ciclo. Una alternativa en la implementación del método es moverse a soluciones vecinas que no empeoren la función objetivo, en cuyo caso no habríamos terminado aún pues $(3, 1, 5, 4, 2)$ cumple con esta propiedad. Y al iterar sobre este nuevo vecino podemos acceder a $(1, 3, 5, 4, 2)$ y $(3, 1, 4, 5, 2)$ que son soluciones que examinamos antes. Si continuamos iterando seguiremos accediendo a soluciones equivalentes y caeremos en un ciclo. Como veremos más adelante, esta es una de las fallas que Búsqueda Tabú trata de eliminar.

3.2. Recocido Simulado

Para explicar la estrategia del método de Recocido Simulado revisemos con un poco más de cuidado la falla de Mejoras Sucesivas; sabemos que el método queda tiene problemas al identificar un óptimo local, ésto se debe a que la exploración se realiza en una sola "dirección"pues sólo se hacen movimientos a vecinos mejores. Si por el contrario la exploración permitiera movimientos a soluciones que empeoren la función objetivo la posibilidad de alejarse del óptimo local sería más probable ya que se permitiría un análisis por varias rutas. Esta es la estrategia de exploración que caracteriza al método de Recocido Simulado. En la industria siderúrgica se lleva a cabo un proceso que inspiró a la aplicación de esta estrategia: el recocido de metales.

Para modificar las propiedades físicas de un metal, como endurecerlo, se recurre al proceso de recocido, el cual consiste en calentar un metal y dejarlo enfriar hasta que llegue a temperatura ambiente; al calentarse, los átomos pueden desplazarse de la posición en la que estaban y cambiar a otro estado de energía que posiblemente permita disminuir la temperatura. Al realizar el proceso de recocido repetidas veces se obtendrá un metal con mejores propiedades y que se ha enfriado a una temperatura menor que la inicial.

El método de Recocido Simulado explora la vecindad de una solución aceptando siempre vecinos mejores y ocasionalmente algunos peores para escapar del óptimo local, aunque estas aceptaciones deben hacerse de manera controlada, de este modo el método podría ir en la dirección correcta y desviarse por tales perturbaciones. La forma de efectuar este control la presenta una de las leyes de la termodinámica empleada en el proceso de recocido la cual establece que a una temperatura T, la probabilidad de tener un cambio de energía δE se puede aproximar de la siguiente manera:

$$\mathbb{P}[\delta] = e^{\left(\frac{-\delta E}{kT}\right)}$$

Donde k es una constante física conocida como constante de Boltzmann. Esta constante no tiene un significado en los problemas de optimización, por lo que la omitiremos . El parámetro T que aparece en esa probabilidad es usado como un medio para regular el Recocido Simulado, conforme pasan las iteraciones se irá disminuyendo el valor de la temperatura T lo que provocará que la probabilidad de aceptación disminuya hasta que no se acepten más soluciones malas, esperando que a este punto se presente una convergencia al óptimo global. En el proceso de recocido este valor es la temperatura a la cuál se calienta el metal antes de dejarlo enfriar lentamenteme; más adelante explicaremos a detalle el proceso para actualizar la temperatura en el método.

Las instrucciones generales que se deben implementar en el recocido simulado son las siguientes.

Recocido Simulado. Es necesario obtener una solución inicial x_0, definir la temperatura inicial $T_0 \geq 0$, el valor de reducción de temperatura $\alpha \in (0, 1)$.

1. Hacer $x_0 = x_{actual}$, hacer $T = T_0$
2. Explorar la vecindad $N(x_{actual})$
   - Si existe $x \in N(x_{actual})$ tal que $c(x)$ es mejor que $c(x_{actual})$ hacer $x_{actual} = x$
   - En caso contrario elegir $x \in N(x_{actual})$ y hacer $\delta = \mid z(x) - z(x_{actual}) \mid$. Generar aleatoriamente un número u de una distribución $U(0,1)$.
     a) Si $u < e^{-\delta/T}$ hacer $x_{actual} = x$
3. Verificar si se debe actualizar T
4. Verificar la condición de paro.
   - Si se ha cumplido terminar.
   - Caso contrario regresar a 2.

No hay especificación alguna sobre qué tan rápido se debe actualizar la temperatura; se ha probado que si desciende de manera muy lenta la convergencia del método será hacia la solución óptima, pero como es de esperarse el tiempo de ejecución del método será demasiado largo, por lo que no es una buena opción. Tampoco se tiene una regla de cada cuando actualizarla ni de cómo calcular la temperatura inicial. Se suele sugerir que la temperatura inicial sea lo suficientemente grande como para permitir aceptar soluciones malas desde el principio.

La manera más común para disminuir la temperatura es por medio de un parámetro $\alpha \in (0,1)$ con el cual calcularemos de manera recursiva los valores de T como sigue:

$$T_i = \alpha T_{i-1}$$

El valor inicial T_0, dados: una solución inicial x y el valor de su función objetivo $z(x)$, se puede calcular como:

$$T_0 = \alpha z(x)$$

Estas posibilidades son sólo sugeridas, pues como mencionamos, no hay regla alguna sobre como elegirlas. Otra alternativa igual de utilizada para calcular la temperatura es usar un parámetro $\beta \in (0,1)$ de forma que:

$$T_i = \frac{T_{i-1}}{1 + \beta T_{i-1}}$$

Respecto al momento de actualizar la temperatura hay dos opciones, la primera es actualizar la temperatura después de un número N de iteraciones, no obstante, en [12] Taha no considera los movimientos de desmejora, por lo que la actualización se hace después de aceptar N soluciones mejores.

A continuación presentaremos la solución del ejemplo que se encuentra al inicio del capítulo, pero esta vez por medio del método de Recocido Simulado.

Igual que con el método de Mejoras Sucesivas $x_0 = 7$ por lo que $T_0 = \alpha f(x_0)$; recordemos que el valor inicial de T, T_0, debe permitirnos aceptar casi cualquier

movimiento de mejora, e ir disminuyendo pero no de manera muy lenta, si tomamos $\alpha = \frac{9}{10}$ por ejemplo, y sabiendo que δ a lo más será de dos tenemos que $e^{-\frac{\delta}{T}}$ será mayor a 0.9237, es decir muy cercano a 1, que es justo lo que deseamos, sin embargo el valor de T_1 será de $\frac{567}{25}$ por lo que $e^{-\frac{\delta}{T}}$ será mayor que 0.9155, en otras palabras la temperatura no disminuye lo suficientemente rápido para descartar movimientos de desmejora al transcurrir las iteraciones. Probando con $\alpha = \frac{7}{10}$ el resultado mejora pues aceptaremos soluciones de desmejora más del 90.29 % de las veces, y después de actualizar la temperatura este valor descenderá a 86.43 y posteriormente a 81.12 lo cual parece ser más razonable tomando en cuenta que el ejemplo no tiene muchas soluciones y que la actualización de la temperatura puede que se haga solo una o dos veces porque la realizaremos después de 3 iteraciones (solo para fines de ejemplificar el método).

El resultado, cuando hacemos una exploración completa de la vecindad es el siguiente:

x	$f(x)$	$N(x)$	Vecino Mejor	N Aleatorio	Vecino elegido	u	e	T
7	28	5,6,8,9	6	–	–	–	–	28(0.7)
6	203	4,5,7,8	–	0.446	5	0.207	0.950	28(0.7)
5	7	3,4,6,7	3	–	–	–	–	28(0.7)
3	149	1,2,4,5	1	–	–	–	–	28(0.49)
1	227	2,3	–	0.642	3	0.280	0.864	28(0.49)
3	149	1,2,4,5	1	–	–	–	–	28(0.49)

Al término del método elegimos como mejor solución a $x = 1$ que coincide con el óptimo, sin embargo, observemos que en la tercera iteración tomamos como mejor solución que la actual a $x = 3$ ya que elegir a 6 nos regresa a una zona ya explorada lo que en un caso desafortunado pudo terminar con el método atrapado por un ciclo; esto ha sido algo que nosotros notamos a simple vista, pero si hubiera sido algo programado la computadora podría elegir al mejor vecino que era 6 y quizá el método habría fallado al quedar atrapado en un óptimo local. El poder identificar un movimiento como no favorecedor requiere mantener cierta "memoria"sobre lo que hemos hecho, técnica en la que se basa el próximo método.

Sobre la exploración aleatoria el método cambia un poco. Si la solución vecina es mejor que la actual se elige sin más, pero si no lo es se prosigue como si no existiera una solución mejor, es decir, se genera un número aleatorio u y se calcula la probabilidad de aceptación con lo que se verifica si se acepta o se sigue explorando la vecindad.

x	$f(x)$	$N(x)$	N Aleatorio	Vecino Elegido	u	e	T
7	28	5,6,8,9	0.568	8	–	–	28(0.7)
8	83	6,7,9,10	0.424	7	0.945	0.950	28(0.7)
7	28	5,6,8,9	0.469	6	–	–	28(0.7)
6	203	4,5,7,8	0.382	5	0.696	0.950	28(0.49)
5	7	3,4,6,7	0.236	3	–	–	28(0.49)
3	149	1,2,4,5	0.507	4	0.286	0.950	28(0.7)

La mejor solución obtenida fue el mínimo local $x = 6$, pero con una elección afortunada pudo haberse encontrado el óptimo con menos recursos.

3.2.1. Recocido Simulado para el problema de Número Cromático

Resolver el problema del número cromático usando Recocido Simulado no es tan obvio como fue con los otros problemas. En este problema cada solución está definida como varios conjuntos de nodos, donde cada uno de ellos representa un color, de modo que definir una vecindad no es tan claro. Nos interesa reducir el número de conjuntos por lo que intercambiar nodos de un conjunto a otro no ayudará mucho a menos que uno de éstos se vuelva vacío. Por otro, lado debemos tomar en cuenta la factibilidad, pues mover un nodo a otro conjunto abre la posibilidad de generar soluciones no factibles. Downsland y Adenzo [8] explican algunas alternativas para definir una vecindad. Nosotros trabajaremos una de ellas en particular.

En el capítulo 2 presentamos este problema y a modo de ejemplo dimos la gráfica de la figura 3.8.

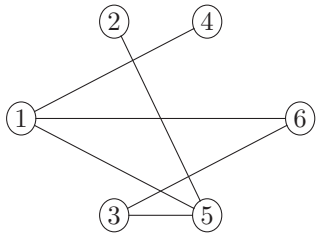

Figura 3.8: Ejemplo del capítulo 2

Una solución factible es colorear cada nodo de un color diferente, que podemos representar como $s_1 = \{(1,1),(2,2),(3,3),(4,4),(5,5),(6,6)\}$ donde cada pareja nos indica el número de nodo y el color respectivamente. Con un Método Glotón encontramos la solución $\tilde{s} = \{(1,1),(2,1),(3,1),(4,2),(5,2),(6,3)\}$ que corresponde a colorear la gráfica con 3 colores. Si consideramos como vecinas a aquellas soluciones a las que podemos acceder mediante la elección aleatoria de un nodo al que le cambiamos el color por otro elegido también de manera aleatoria, tendremos una manera muy sencilla de generar vecinos, sin embargo, como mencionamos antes no todos serán factibles. Por ejemplo, un vecino de s_1 es $\{(1,4),(2,2),(3,3),(4,4),(5,5),(6,6)\}$, que tiene 5 colores unicamente (el color 1 ya no aparece), no obstante, al ser 1 y 4 adyacentes no es factible.

Aún con este problema podemos utilizar esta vecindad si corregimos esto con la función objetivo. Dijimos que una solución es una serie de conjuntos que representan a un color, de modo que entre más grande sea la cardinalidad de éstos serán menos conjuntos (como es el caso del vecino que dimos, que tiene solo 5 colores al tener un conjunto de mayor cardinalidad, el asociado al color 4). El problema es que debemos penalizar a las soluciones que no sean factibles, para lo que contaremos el número de arcos "malos".

Entonces, la función objetivo deberá incentivar el aumento en la cardinalidad de los conjuntos que representan a cada color y penalizar el número de arcos malos. La función objetivo propuesta es la siguiente:

$$z(s) = \sum_{j=1}^{k} (|C_j|)^2 - 2\sum_{j=1}^{k} |C_j||A_j|$$

Donde k es el número de conjuntos que representan a cada color, C_j es el conjunto asociado al j-ésimo color y A_j es el conjunto de arcos que conectan a nodos coloreados con el j-ésimo color.

De esta manera el valor de la función objetivo en s_1 se calcula como sigue:

$$(1^2 + 1^2 + 1^2 + 1^2 + 1^2 + 1^2) - 2(1 * 0 + 1 * 0 + 1 * 0 + 1 * 0 + 1 * 0 + 1 * 0) = 6$$

Y la función, cuando la solución es $\tilde{s}$, tiene un valor de 14 calculada de la siguiente manera:

$$(3^2 + 2^2 + 1^2) - 2(3 * 0 + 2 * 0 + 1 * 0) = 14$$

Y si tenemos la solución $\{(1,4),(2,2),(3,3),(4,4),(5,5),(6,6)\}$ obtenemos un valor de 4 en su función objetivo porque tiene un arco malo $((1^2 + 1^2 + 1^2 + 2^2 + 1^1) - 2(1 * 0 + 1 * 0 + 1 * 0 + 2 * 2 + 1 * 0))$. Con lo que podemos ver que la penalización de los arcos malos es suficiente para descartar soluciones que aparentan ser buenas por colorear la gráfica con un número menor. También notemos que la solución encontrada con el glotón es la mejor de todas y su valor de z es el más grande, de modo que el problema consiste en maximizar z.

Otro punto a considerar es el número de vecinos que se generan. Dijimos que se selecciona un número y un color al azar de modo que no deben ser más de $n*k$ vecinos, donde k es el número de colores que tiene la solución y n el número de nodos. Sin embargo, para que sean considerados vecinos deben ser distintos a la solución actual por lo que de esos $n * k$ k soluciones no serán vecinas (por ejemplo elegir al nodo 3 y el color 3 cuando s_1 es la solución actual). Así que el número de vecinos es $n * (k - 1)$.

Si la gráfica de arriba es nuestro problema, y la solución tiene 6 colores tenemos un total de 30 vecinos, y si tenemos 3 colores como en el caso de la solución encontrada con el glotón tendremos 12. Si realizamos unas 5 iteraciones habremos revisado más soluciones que las necesarias en caso de revisar una por una las soluciones factibles. Esto es un buen argumento para justificar el uso de una exploración aleatoria de la vecindad. En este caso, aceptaríamos siempre a los vecinos mejores y con cierta probabilidad a aquellos que no lo sean.

Antes de describir el método para este problema en específico es importante hacer una observación más. El hecho de que una solución sea mejor que otra no garantiza que sea factible. De modo que al actualizar la mejor solución sólo

deberemos hacerlo si la factibilidad está garantizada. Es por esto que el método queda como sigue:

Recocido Simulado para el problema del número cromático.
Parámetros iniciales: Solución inicial s_0, Temperatura inicial T_0, tasa de decremento de temperatura α, Número de soluciones antes de cambiar la temperatura N, número de iteraciones T^{max}

1. Hacer $s_{actual} = s_0$, $T = T_0$, $s_{mejor} = s_0$, $n = 0$ y $t = 0$
2. Construir un vecino (s_{vecino}) eligiendo un nodo y un color de manera aleatoria. Reemplazamos el color del nodo elegido con el color elegido. Si la elección aleatoria no genera un vecino se debe repetir hasta que así suceda.
3. Calcular $z(s_{vecino})$ y $z(s_{actual})$ con la fórmula:
   $z(s) = \sum_{j=1}^{k}(|C_j|)^2 - 2\sum_{j=1}^{k}|C_j||A_j|$
   - Si $z(s_{vecino}) > z(s_{actual})$ (Recordemos que tratamos el problema como uno de maximización) hacer $s_{actual} = s_{vecino}$
   - En otro caso generar un número aleatorio $R \in [0, 1]$.
     - Si $R < e^{-\frac{-|z(s_{actual})-z(s_{vecino})|}{T}}$ hacer $s_{actual} = s_{vecino}$
     - En otro caso ir a 4
4. Hacer $n = n + 1$. Si $n = N$ actualizar la temperatura $T = \alpha T$ y hacer $n = 0$
5. Si $s_{actual} \in F_P$ y mejor, hacer $s_{mejor} = s_{actual}$
6. Verificar si $t = T^{max}$
   - De ser así x_{mejor} es la solución encontrada
   - En caso contrario hacer $t = t + 1$ y volver a 2

Para el ejemplo probaremos con 5 iteraciones, actualizando la temperatura cada 3 (no solo cada dos movimientos de mejora porque 5 iteraciones quizá no alcanzarían), la temperatura inicial será de 4.2 y la disminución de ésta sera de 0.3 ($\alpha = 0.7$). La solución actual será la de asignar un color a cada nodo.

Primera Iteración

Paso 1. Iniciamos el método asignando los valores iniciales. $s_{mejor} = s_{actual} = s_0 = \{(1,1)(2,2)(3,3)(4,4)(5,5)(6,6)\}$, $T = 4.2$ y $n = t = 0$

Paso 2. El vecino lo encontramos modificando el color del nodo 4 al color 2. $s_{vecino} = \{(1,1)(2,2)(3,3)(4,2)(5,5)(6,6)\}$

Paso 3. Recordemos que antes de comenzar este ejemplo calculamos el valor de la función objetivo en s_0 obteniendo el valor de 6, así que solo falta el de s_{vecino}

$z(s_{vecino}) = (1^2 + 2^2 + 1^2 + 1^2 + 1^2) - 2(1*0 + 2*0 + 1*0 + 1*0 + 1*0) = 8$

Observemos que esta solución es mejor además de ser factible por lo que automáticamente la elegimos como solución actual.
$s_{actual} = \{(1,1)(2,2)(3,3)(4,2)(5,5)(6,6)\}$

Paso 4. $n = 1$ de modo que no actualizamos la temperatura

Paso 5. Como la nueva solución actual es factible y es mejor también debemos actualizar este valor. $s_{mejor} = \{(1,1)(2,2)(3,3)(4,2)(5,5)(6,6)\}$

Paso 6. Hagamos $t = 1$ y volvemos a 2

Segunda Iteración

Paso 2. El nuevo vecino es $\{(1,1)(2,2)(3,1)(4,2)(5,5)(6,6)\}$

Paso 3. Calculamos el valor de z del nuevo vecino $z(s_{vecino}) = 10$ que nuevamente es mejor y por lo tanto lo seleccionamos.

Paso 4. $n = 2$ y pasamos al paso 5

Paso 5. También hacemos esta solución igual a la mejor pues también es factible

Paso 6. Hagamos $t = 2$ y volvemos al paso 2

Tercera Iteración

Paso 2. El siguiente vecino es $\{(1,6)(2,2)(3,1)(4,2)(5,5)(6,6)\}$

Paso 3. El valor de la función objetivo de esta solución es 6 que resulta menor a la que teníamos antes. Por lo tanto debemos generar un número aleatorio R y la probabilidad de elección.

- $R = 0.317900$
- $Probabilidad = e^{-\frac{|10-6|}{4.2}} = 0.385821$

Como R es menor elegimos a s_{vecino} como actual solución.

Paso 4. $n = 3$ y pasamos al paso 5

Paso 5. Como esta solución implica colorear a 1 y 6 del mismo color no está en F_P así que la mejor solución permanece igual.

Paso 6. Hagamos $t = 3$ y volvemos al paso 2

Cuarta Iteración

Paso 2. El cuarto vecino generado es $s_{vecino} = \{(1,6)(2,2)(3,1)(4,6)(5,5)(6,6)\}$

Paso 3. La función objetivo de esta solución es igual a cero que es el peor resultado encontrado hasta ahora. Debemos ver si la elegimos o no. El número aleatorio $R = 0.892431$, y la probabilidad de aceptar es de 0.239651 que es muy baja pues está muy alejada de esta solución

Paso 4. El valor de $n = 4$ ahora de modo que la temperatura bajará a 2.94 y hacemos $n = 0$

Paso 5. La solución no será actualizada porque ni es factible ni es mejor

Paso 6. Hagamos $t = 4$ y volvemos al paso 2

Quinta Iteración

Paso 2. El último vecino es $\{(1,6)(2,5)(3,1)(4,2)(5,5)(6,6)\}$

Paso 3. El valor de la función objetivo es 2 por lo que no la aceptamos inmediatamente. Al ser peor ya sabemos quién es mejor, no obstante, debemos terminar de iterar. El número aleatorio es $R = 0.408081$ y la probabilidad de aceptar esta solución es 0.256521 de modo que no lo hacemos

Paso 4. El último valor que n tendrá es el de 1 que no alcanza para actualizar la temperatura de nuevo

Paso 5. No actualizamos a la mejor solución

Paso 6. Hagamos $t = 5$ y como ya se ha alcanzado terminamos con una solución de 4 colores

Podemos notar que en este caso la solución encontrada fue peor que la encontrada por un método glotón, sin embargo puede deberse a que tomamos un número pequeño de iteraciones y que la solución inicial era la más mala posible, si bien hubiéramos iniciado con la solución obtenida por el método glotón sería posible encontrar la solución óptima de 2 colores (realizamos este ejercicio con el código diseñado para este problema y ese resultado se obtuvo la mayoría de las veces).

3.2.2. Recocido Simulado para el problema Lineal Entero

Hemos mencionado que la principal falla del método de Mejoras Sucesivas es el estancamiento en un óptimo local, esta falla se presenta con mayor frecuencia en estos problemas. Por ejemplo, el siguiente caso problema de la mochila:

$$
\begin{aligned}
&Max\ z = 10x_1 + 3x_2 \\
&\text{s.a} \\
&20x_1 + 3x_2 \leq 20 \\
&x_i \in \{0, 1\}
\end{aligned}
$$

La solución óptima es $(1, 0)$. Si usamos como solución inicial la obtenida por un método glotón $(0, 1)$ sólo podemos llegar al vecino $(0, 0)$ que es peor y habríamos terminado. No obstante, usando Recocido Simulado sí nos podríamos mover a $(0, 0)$ de donde podríamos pasar a las solución óptima. Lo que queremos decir es que el desempeño de este método es más prometedor para este problema. Además, la manera de calcular el valor de z conociendo el del vecino puede ser sencilla.

Para el ejemplo revisaremos la vecindad completa utilizando las siguientes instrucciones:

Recocido Simulado para el Problema Lineal Entero (Maximización).
El método necesita una solución inicial x_0, una temperatura inicial $T_0 = \alpha z(x_0)$, un valor α y el número de soluciones antes de actualizar la temperatura N.

1. Hacer $x_{mejor} = x_{actual} = x_0$, $T = T_0$, $n = t = 0$
2. Construir un vecino factible de la siguiente manera: Elegir un número $p \in \{-k, -(k-1), ..., 0, ..., k-1, k\}$ y un entero $j \in \{1, 2, ..., n\}$, de esta manera el siguiente vecino queda como:

   $$x_{vecino} = (x_1, x_2, ..., x_j + p, ..., x_n)$$

   Con $x_1, x_2, ..., x_n$ los valores de la solución actual.
3. Calcular el valor $c_j p$
   - Si es mayor que cero hacemos $x_{actual} = x_{vecino}$
   - En otro caso elegimos al vecino con la siguiente probabilidad[3]: $e^{-\frac{|c_j p|}{T}}$
4. Actualizamos el valor de $n = n + 1$ Si $n = N$ actualizar la temperatura $T = \alpha T$ y hacer $n = 0$
5. Si x_{actual} es mejor, hacer $s_{mejor} = s_{actual}$
6. Verificar si $t = T^{max}$
   - De ser así x_{mejor} es la solución encontrada
   - En caso contrario hacer $t = t + 1$ y volver a 2

Resolveremos el siguiente ejemplo:

$$
\begin{aligned}
&Max z = 3x_1 + 2x_2 + x_3 \\
&\text{s.a} \\
&-x_1 + 2x_2 + x_3 \leq 4 \\
&3x_1 + 2x_2 \leq 14 \\
&x_1 - x_2 \leq 3 \\
&x_i \in \mathbb{Z}^+
\end{aligned}
$$

[3]El algoritmo trabaja con la fórmula $|z(x_{actual}) - z(x_{vecino})|$ que es la diferencia entre las funciones objetivo de las soluciones, que resulta ser exactamente en valor $c_j p$ de modo que no necesitamos hacer el cálculo de nuevo

Como solución inicial tomaremos una solución al estilo glotón. La variable que aporta más valor a la función objetivo es x_1 y su valor máximo es 3 si suponemos que las demás variables son iguales a cero. La segunda variable es x_2 que con el valor de $x_1 = 3$ a lo más puede valer 2, y finalmente la variable x_3 puede valer hasta 3 con los otros valores dados, así tenemos que $x_0 = (3, 2, 3)$. Su valor objetivo es igual a $9 + 4 + 5 = 18$, de modo que el valor de $T_0 = 12.6$ si tomamos un alfa de 0.7 como en el ejemplo anterior. Consideraremos una actualización de temperatura después de 2 movimientos de mejora. Y vamos a realizar 5 iteraciones. La vecindad la definiremos con un valor de $k = 3$.

Primera Iteración

Paso 1. Iniciamos los valores y hacemos $x_{mejor} = x_{actual} = x_0$, $T = 12.6$ y $n = t = 0$

Paso 2. El vecino será $(3, 2, 0)$ que se construye restando 3 unidades a la variable x_3.

Paso 3. Ahora calculamos $c_jp = -1 * 3 = -3$. Como es un valor negativo sabemos que es una solución peor, de modo que revisamos si la elegimos o no. El número aleatorio R será 0.805196 y la probabilidad de elección es 0.788127 que es menor, así que no lo elegimos

Paso 4. Como no hubo movimiento de mejora el valor de n sigue siendo 0

Paso 5. No es necesario actualizar puesto que no cambiamos de solución

Paso 6. Hacemos $t = 1$ y pasamos a 2

Segunda Iteración

Paso 2. Probaremos con el vecino $(3, 0, 3)$ que consiste en restarle 2 unidades a x_2

Paso 3. El valor de $c_jp = -4$. Solución que nuevamente es peor. El nuevo número aleatorio es 0.243121 y la probabilidad es de 0.727995 por lo que sí la aceptamos esta vez

Paso 4. n sigue siendo 0 pues aunque nos movimos de solución no hubo mejora.

Paso 5. Tampoco cambiamos el valor de la mejor solución.

Paso 6. Hacemos $t = 2$ y pasamos a la siguiente iteración en el paso 2.

Tercera Iteración

Paso 2. El siguiente vecino será $(4, 0, 3)$ elegido al aumentar en 1 unidad la variable 2.

Paso 3. El producto $c_j p = 2$ que es positivo por lo que lo elegimos automáticamente. $x_{actual} = (3, 1, 3)$.

Paso 4. Hacemos $n = 1$ pues ha ocurrido nuestro primer movimiento de mejora y pasamos a 5.

Paso 5. Aunque fue una mejora con respecto a la solución actual no lo fue con respecto a la mejor por lo que seguimos sin actualizar a la solución mejor.

Paso 6. Hacemos $t = 3$ y pasamos a 2.

Cuarta Iteración

Paso 2. La siguiente solución será $(4, 1, 3)$ que es una unidad mayor que la anterior con respecto a x_1.

Paso 3. La diferencia con respecto a la solución actual es de 3, esto quiere decir que también se acepta, entonces $s_{actual} = (4, 1, 3)$.

Paso 4. Tenemos nuestro segundo movimiento de mejora y $n = 2$.

Paso 5. Esta nueva solución sí es mejor. Recordemos que la solución anterior tenía un valor de z de 14 y como el incremento fue de 3 esta nueva lo tiene de 17. La mejor solución que tenemos era la inicial con un valor de 16. Por lo tanto $z_{mejor} = (4, 1, 3)$.

Paso 6. Hacemos $t = 4$ y pasamos a 2.

Quinta Iteración

Paso 2. La siguiente solución es $(2, 1, 3)$ Sin hacer cuentas podemos notar que no será un movimiento de mejora pues solo le restamos valor a una variable.

Paso 3. Este vecino tiene una diferencia de 6 unidades mejor en su función objetivo con respecto a la solución actual ($c_j p = -6$). El nuevo valor de R es 0.361616 y $e^{-\frac{6}{12.6}} = 0.621145$ así que sí lo elegimos como solución actual.

Paso 4. Este movimiento no fue de mejora. Razón por la cual terminaremos sin haber cambiado la temperatura ni una sola vez.

Paso 6. Hacemos $t = 5$ y terminamos el proceso. La mejor solución encontrada fue: $(4, 1, 3)$.

Esta implementación fue rápida y hubo una mejora, aunque no considerable, podemos mencionar dos inconvenientes que se presentan al aplicar el método. El primero es que el valor de k quizá fue grande pues aunque no lo apuntamos, se obtuvieron varios vecinos no factibles que tuvimos que descartar. Por otro lado el número de iteraciones fue pequeño de modo que varios movimientos disminuyen la función objetivo sin que permitiéramos al método empezar los de mejora.

3.3. Búsqueda Tabú

La estrategia que utiliza el método de Búsqueda Tabú es un poco más complicada en comparación con otros métodos, esto se debe a que su principal propósito es realizar exploraciones mejores en cada iteración aprovechando los resultados obtenidos en las anteriores. Para lo cual se basa en un proceso de memoria que le permite "prohibir"la exploración de soluciones con características que se recuerden como malas o poco prometedoras. La importancia del proceso de memoria está en la idea de que obtener soluciones malas mediante una estrategia inteligente es mejor que obtener resultados buenos pero obtenidos al azar, lo que significa, que aún cuando una vecindad no proporcione una mejor solución beneficiará al método proporcionando información sobre las características malas de las soluciones.

Dicho proceso de memoria puede estar basado hasta en cuatro aspectos sobre la exploración de la vecindad:

- Si el aspecto a analizar es el número de veces que se ha realizado un movimiento de intercambio se dice que la memoria está basada en lo frecuente.
- Si se revisa que el movimiento no sea el mismo que hace pocas iteraciones la memoria estará basada en lo reciente.
- Si se considera qué tan buena es la solución obtenida con respecto a las demás está basada en la calidad.
- Si se toma en cuenta el que la solución permita obtener más soluciones mejores se dice que está basada en la influencia.

Sin embargo, generalmente no es necesario basarlo en los cuatro aspectos, es suficiente que la memoria se concentre en los primeros dos aspectos. De la misma manera, la mayoría de las características del método pueden modificarse tanto como se considere necesario para obtener el mejor resultado posible, lo que lo convierte en un método altamente personalizable a diferencia de los dos métodos de búsqueda tabú y mejoras sucesivas.

Una vez que se tiene proceso de memoria se identificará a los elementos de la vecindad $N(x)$ que no serán explorados por ser malos candidatos y se marcarán como elementos tabú (es decir, serán los elementos prohibidos) durante

un determinado número de iteraciones. Este número puede ser estático, o lo que es lo mismo, el periodo tabú de un movimiento es el mismo en cualquier momento y está definido de antemano al inicio el método; por otro lado, puede ser dinámico, es decir, que irá variando entre una cota inferior y una superior en cada movimiento de intercambio realizado y puede ser de dos maneras: aleatorio o sistemático, en el caso sistemático la duración del periodo tabú está determinada por alguna característica del movimiento realizado.

Reducir la vecindad trae la misma ventaja que en los otros heurísticos, el método se realiza con mayor velocidad. A cambio, como siempre, de la posible pérdida de una buena solución por causa de un movimiento prohibido. Por lo que la condición tabú de un movimiento casi nunca se hace estrictamente inviolable, si el realizar un movimiento tabú implicara un resultado mejor a los obtenidos anteriormente se puede hacer una excepción y realizar el movimiento. A esta consideración se le conoce como criterio de aspiración, su objetivo es el de mejorar el método al no excluir soluciones buenas por malas identificaciones de la memoria; así que puede definirse más estricto o relajado según convenga, por ejemplo, se puede aplicar el criterio de aspiración unicamente cuando los elementos no tabú proporcionen soluciones mejores, o simplemente no aplicar ningún criterio y de esta manera disminuir el esfuerzo del método.

Otra característica de Búsqueda Tabú es que a diferencia de los métodos anteriores, éste se mueve a la mejor solución de la vecindad, aún cuando no sea mejor que la solución actual. Por lo que al igual que con Recocido Simulado es necesario definir un número de iteraciones o algún otro criterio de paro.

De manera general el método de Búsqueda Tabú es el siguiente:

Búsqueda Tabú con exploración completa. Iniciamos con una solución inicial x_0 y definimos la duración del periodo tabú como n.

1. Hacer $x_{actual} = x_0$ y marcar a x_0 como tabú.
2. Obtener los vecinos de x_{actual} que denotaremos como $N(x_{actual})$ y obtener la vecindad de elementos no tabú $N'(a_{actual})$ de la siguiente manera: $N'(a_{actual}) = N(x_{actual}) - L_t$ con L_t el conjunto de elementos marcados como tabú hasta ese momento.
3. Explorar $N(x_{actual})$
   - Si existe $x \in N(x_{actual}) \cap L_t$ tal que $c(x)$ es mejor que cualquier resultado obtenido a la fecha hacer $x_{actual} = x$.
   - Caso contrario exploramos $N'(x_{actual})$ y elegimos x tal que $c(x) \geq c(x')\ \forall x' \in N'(x_{actual})$ y hacemos $x_{actual} = x$.
4. Actualizamos la lista de elementos tabú. Actualizamos la condición de término.
5. Verificar la condición de término:
   - Si no se ha cumplido regresar a 2.
   - Caso contrario terminamos el método.

Para el ejemplo tomaremos como criterio de paro el realizar 4 iteraciones, la solución inicial $x_0 = 7$, una permanencia tabú de un periodo y una vecindad $N(x) = \{x-2, x-1, x, x+1, x+2\}$.

Iteración	x_{actual}	$N(x_{actual})$	Elementos Tabú	$N'(x_{actual})$	$f(x)$	Mejor solución
0	7	5,6,8,9	7	5,6,8,9	28	6
1	6	4,5,7,8	6,7	4,5,8	203	4
2	4	2,3,5,6	4,6	2,3,5	113	3
3	3	1,2,4,5	3,4	1,2,5	149	1
4	1	2,3	1,3	2	227	2

Después de terminar las 4 iteraciones seleccionamos la mejor solución obtenida hasta ahora que resulta ser el óptimo en este caso ($x = 1$). También cabe destacar que se logró escapar del mínimo local donde el método de mejoras sucesivas quedó atrapado.

Ahora revisaremos el método para el caso donde la vecindad no es explorada del todo. En donde lo que haremos es modificar el paso 3 del método anterior para cambiarlo por la instrucción 3'.

3'. Explorar $N'(x_{actual})$ y elegir de manera aleatoria a un x para hacer $x_{actual} = x$.

Aquí podemos ver que la condición de tabú sí es estricta, pues no exploraremos a estos aunque sean vecinos. A favor de esta modificación podemos decir que se incrementará el número de zonas exploradas. Siguiendo las instrucciones descritas arriba obtenemos los resultados que se muestran en la siguiente tabla:

Iteración	x_{actual}	$N(x_{actual})$	Elementos Tabú	$N'(x_{actual})$	$f(x)$	Num. Aleatorio
0	7	5,6,8,9	7	5,6,8,9	28	0.945
1	9	7,8,10,11	7,9	8,10,11	111	0.328
2	8	6,7,9,10	8,9	6,7,10	83	0.181
3	6	4,5,7,8	8,6	4,5,7,9	203	0.702
4	7	5,6,8,9	6,7	5,8,9	28	–

Esta exploración pierde parte del propósito de analizar inteligentemente la vecindad, pues en realidad la exploración se limita a analizar elementos al azar sin tratar de repetir

Recordemos que el criterio de aspiración se refiere al análisis de elementos tabú y elegirlo sólo en caso de que proporcione una solución mejor a cualquiera de las encontradas hasta ese momento. Para este problema en particular el criterio podría ignorarse y se obtendría el mismo resultado en el primer ejemplo. De hecho, en el segundo caso no se consideró el criterio. Dependiendo del problema, podemos modificar algunos elementos del método general, ya sea para obtener mejores resultados, o para ganar velocidad durante el análisis. Por ejemplo, supongamos que en el ejemplo nos interesara saber si existe un valor de x para el cual la función alcanza un valor superior a 100, el criterio de paro pudo haberse modificado de manera que el método se detenga si se realizan cuatro iteraciones o si se encuentra algún valor con las características deseadas, lo que ocurra primero.

Al ser un problema pequeño, nos bastó con utilizar una memoria basada en lo *reciente*; es decir, se prohíbe la exploración de elementos que han sido analizados hace poco. Pero con problemas de mayor tamaño bien podría no ser suficiente; si además se considerara como tabú aquellos elementos que han sido

examinados k veces o más, aún si esto no ha sido recientemente, o al menos considerarlos menos atractivos para su revisión, podría obtenerse un resultado mejor, esta memoria es la basada en lo *frecuente*. Pese a que no es el objetivo, el método de Búsqueda Tabú reduce su probabilidad de caer en ciclos debido a estas dos memorias; lo que sí pretende el método, es la reducción de la vecindad, que resulta de gran utilidad especialmente para el caso en que se explora la vecindad por completo.

Sin embargo, reducir la vecindad tiene sus desventajas, nuevamente se sacrifica rapidez por precisión, he ahí la razón del criterio de aspiración. Como ya dijimos, en casos donde movimientos que han sido prohibidos sí pueden obtener un mejor resultado éste entra en acción, compensando las prohibiciones con las memorias de corto y mediano plazo. Otra técnica para mejorar la calidad de las exploración en añadir a la vecindad *soluciones élite* que son aquellas que posean alguna característica que se piensa, podría producir soluciones mejores.

Por último, pese a que ese no es su objetivo, el método de Búsqueda Tabú tiene otra ventaja ante los métodos de Recocido Simulado y Mejoras Sucesivas y es que el mismo proceso de memoria hace menos probable que en ciclos, a diferencia de lo que ocurre con el método de Mejoras Sucesivas. La prevención del ciclado va de la mano con las memorias, sin embargo, también tiene una relación con la duración de una solución o movimiento como tabú. Si un elemento permanece prohibido por un tiempo adecuado, bastará con la memoria de corto plazo para no caer en ciclos. La duración T de un elemento como tabú puede ser estática (como se hizo en nuestro ejemplo) o dinámica, es decir, que varíe en cada iteración entre dos cotas t_{min} y t_{max}. A su vez, esta variación puede ser de dos maneras:

- *Aleatoria*. Si seleccionamos un tiempo t de manera aleatoria entre t_{min} y t_{max}.
- *Sistemática*. Si la duración t se elige entre t_{min} y t_{max} con un criterio más elaborado como alguna característica del elemento.

Así como nos es posible prohibir la revisión de malas decisiones, debería ser recomendable premiar las elecciones buenas, por ejemplo los movimientos que produzcan un mayor aumento en la función objetivo, o las soluciones que tengan una mejor calidad, por lo que también está la posibilidad de modificar la función objetivo c por una c' que premie la calidad y la influencia de una solución.

3.3.1. Búsqueda Tabú para el problema de Secuencia de Tareas

Para aplicar este método utilizaremos lo que vimos cuando lo resolvimos con el método de Mejoras Sucesivas, sabemos que además de explorar la vecindad debemos crear una lista tabú L_T donde recordaremos los movimientos que hemos hecho. Para utilizar esta lista tenemos dos opciones, comentamos que un

movimiento puede ser tabú sin que esto sea definitivo, ya que si un movimiento tabú produce una solución mejor se puede omitir esta condición; pero si lo que hacemos es una exploración aleatoria podríamos hacer esta condición totalmente restrictiva pues no tenemos idea del valor de las demás soluciones (tanto tabú como no tabú).

En [12] Taha utiliza una exploración aleatoria, por lo tanto nosotros ejemplificaremos la exploración completa. Recordemos que cuando usamos el método de Mejoras Sucesivas hubo un inconveniente con la última iteración pues encontramos soluciones con una función objetivo igual y que nos pudo dejar en un ciclo. Se supone que Búsqueda Tabú evitará ese fallo pues marcaremos como tabú el movimiento.

Durante la descripción del método hablaremos de la lista tabú, sin embargo, debemos tener una manera de recordar cuánto tiempo lleva marcado el movimiento. A modo de propuesta nosotros trabajaremos con una pareja. Si tenemos la solución $(1, 2, 3, 4, 5)$ y pasamos a la $(1, 3, 2, 4, 5)$, el elemento que se añade a la lista será el intercambio $(2, 3)$ y mantendrá su condición por n iteraciones. Por lo tanto, lo añadiremos a la lista como $\{(2, 3), n\}$

Las instrucciones son las siguientes:

Búsqueda Tabú con exploración completa.

1. Hacer $s_{mejor} = s = s_0$, hacer $L_T = \emptyset$
2. Encontrar la vecindad $N(s)$ y elegir al mejor vecino, sea s' ese vecino
3. Para todos los elementos $((i, j), n)$ hacer $n = n - 1$. Si algún $n = 0$ retirarlo del conjunto. Agregar el intercambio de s a s' a la lista tabú por n iteraciones
4. Hacer $s = s'$ si $z(s)$ mejor que $z(s_{mejor})$ hacer $s_{mejor} = s$
5. Hacer $t = t + 1$
   - Si $t = T^{max}$ terminar, s_{mejor} es la mejor solución
   - En caso contrario regresar a 2

Resolveremos el mismo ejemplo que con el método de Mejoras Sucesivas, la duración del periodo tabú será de dos iteraciones y el número de iteraciones en total será de 5, trataremos de ver si el desempeño es mejor que con el método de Mejoras Sucesivas. La solución inicial será la misma $s_0 = (1, 2, 3, 4, 5)$, $z(s_0) = 1,734$. Anotaremos nuevamente los datos del problema que se muestran en la siguiente tabla:

Tarea	Tiempo de realización (días)	Fecha límite (día)	Penalización (por día)
1	10	24	14
2	28	58	16
3	9	21	9
4	15	22	10
5	25	32	20

Y la aplicación del método para resolverlo se muestra a continuación:

Primera Iteración

Paso 1. Iniciamos las variables $s_{mejor} = s = s_0 = (1, 2, 3, 4, 5)$, $L_T = \emptyset$

Paso 2. $N(s) = \{(2, 1, 3, 4, 5), (1, 3, 2, 4, 5), (1, 2, 4, 3, 5), (1, 2, 3, 5, 4)\}$. El mejor vecino es $(1, 3, 2, 4, 5)$ que disminuye la función objetivo en 234 unidades a z

Paso 3. Como la lista tabú está vacía sólo añadimos al movimiento $(2, 3)$ por 2 periodos.

Paso 4. Cambiamos el valor de s por este vecino y como mejoró la función objetivo también la elegimos como mejor solución.

Paso 5. Sea $t = 1$ y volvemos a 2

Segunda Iteración

Paso 2. La nueva vecindad es $N(s) = \{(3, 1, 2, 4, 5), (1, 2, 3, 4, 5), (1, 3, 4, 2, 5), (1, 3, 2, 5, 4)\}$, de donde debemos notar que $(1, 2, 3, 4, 5)$ incluye el movimiento tabú por lo que debe ser descartado sin siquiera calcular el costo de la penalización. El mejor vecino ahora es $s' = (1, 3, 4, 2, 5)$

Paso 3. La lista tabú incluye a $(2, 3)$ durante un periodo más, añadimos a $(2, 4)$ durante 2 periodos

Paso 4. $s = (1, 3, 4, 2, 5)$ que nuevamente reemplazará a la mejor solución $s_{mejor} = (1, 3, 4, 2, 5)$

Paso 5. Sea $t = 2$ y volvemos a 2

Tercera Iteración

Paso 2. La nueva vecindad incluye a los elementos $\{(3, 1, 4, 2, 5)$,$(1, 4, 3, 2, 5)$,$(1, 3, 2, 4, 5)$, $(1, 3, 4, 5, 2)\}$, donde sólo un elemento es tabú:$(1, 3, 2, 4, 5)$, podemos ver que de las demás $(1, 3, 4, 5, 2)$ tiene una disminución de -160

Paso 3. Retiramos al primer elemento de la lista tabú quedando los movimientos $(2, 4)$ y $(2, 5)$ por 1 y 2 periodos respectivamente

Paso 4. Hacemos $s = (1, 3, 4, 5, 2)$ que nuevamente es mejor que nuestra solución pues tiene una función objetivo de 1124

Paso 5. Sea $t = 3$ y volvemos a 2

Cuarta Iteración

Paso 2. Los vecinos esta vez son $(3,1,4,5,2)$,$(1,4,3,5,2)$,$(1,3,5,4,2)$ y $(1,3,4,2,5)$, tenemos sólo un elemento tabú: $(1,3,4,2,5)$. La mejor es $(1,3,5,4,2)$ que no es tabú por lo tanto la elegimos.

Paso 3. Actualizamos la lista tabú retirando a $(2,4)$ y añadiendo a $(4,5)$

Paso 4. Cambiamos el valor de la nueva solución y de la mejor solución: $s = (1,3,5,4,2) = s_{mejor}$

Paso 5. Sea $t = 4$ y volvemos a 2

Quinta Iteración

Paso 2. La última vecindad construida es $\{(3,1,5,4,2)$,$(1,5,3,4,2)$,$(1,3,4,5,2)$, $(1,3,5,2,4)\}$ de donde la mejor solución encontrada es $(1,3,5,2,4)$

Paso 3. Los últimos movimientos tabú serán $(4,5)$ y $(2,4)$

Paso 4. Igualmente reemplazamos la mejor solución por esta nueva

Paso 5. Sea $t = 5$ y terminamos

El resultado encontrado fue el mismo, que con mejoras sucesivas, esto se debió a que en este ejemplo en particular siempre hubo vecinos mejores, de modo que el método nunca avanzó a vecinos que empeoraran la función objetivo. Eligiendo una solución de manera aleatoria el resultado podría ser distinto.

Capítulo 4

Explicación del programa de cómputo

Hacer programas para estos métodos no es una cuestión de pereza, si bien son un apoyo para resolver problemas en menos tiempo, tenemos otras ventajas como son el resolver instancias de mayor tamaño, así como probar distintos escenarios; por ejemplo cambiar valores de los parámetros o en el caso de los métodos de búsqueda local probar con diferentes soluciones iniciales (si es que disponemos de ellas). Por esta razón incluiremos en este capítulo algunos códigos que utilizamos para resolver unos de los ejemplos mencionados en los anteriores.

Los códigos, que están en el anexo al final de este trabajo, fueron escritos en lenguaje C#, por lo que pueden reproducirse con programas como Visual Studio o Sharp Develop[1]. Para no entrar tanto en los detalles del lenguaje únicamente daremos una explicación del código.

4.1. Explicación del programa para el problema de la mochila usando un método glotón

Al igual que la descripción que hicimos del método en el capítulo 2, la programación de este método fue la más sencilla de hacer. Básicamente podemos resumir el código en 5 partes:

- Extraer la información de un archivo de texto
- Calcular la función de selección para cada variable x_i, i = 1,...,n
- Ordenar las variables según el atractivo que tengan (definido por la función de selección que calculamos antes)
- Asignar el valor de 0 o 1 para cada x_i, i = 1,...,n

[1]Nosotros los probamos en la versión 2013 de VS Express y la 4.4 de Sharp Develop

- Recuperar la solución obtenida y guardarla en un archivo de texto

El archivo de texto que utilizamos es de esos que tienen terminación *txt*. Para extraer la información el programa necesita que esté guardado en la misma ruta que el archivo ejecutable bajo el nombre de "Datos_GM". La primera línea tener el valor de la capacidad de la mochila; lo siguiente que se necesita es el peso de los artículos asociado a cada variable x_i separados por coma, de la misma manera hacemos lo mismo para el beneficio de cada artículo.

Para el ejemplo que dimos en el capítulo 2 el archivo debe tener las siguientes líneas:

```
100
10,20,30,40,50
30,28,50,18,59
```

Observemos que, dado que cada variable x_i representa la cantidad de artículo i que meteremos a la mochila podemos asociarle la información de peso, beneficio y la función de selección. De modo que para hablar del peso del artículo i lo anotaremos como $x_i.peso$, y de manera similar con los demás valores. Cuando le asignemos valor a la variable la denotaremos como x_i únicamente pero en el código aparece como $x_i.valor$.
El diagrama de flujo 4.1 ilustra cómo trabaja el programa.

Una parte de gran importancia en el programa es ordenar las variables, una vez que tenemos el valor de la función de selección debemos buscar una manera para que en el siguiente ciclo "for" la variable x_1 sea la mejor según éste nuevo valor. De esta manera se le asignará valor de 1 a las variables más atractivas primero. Pero también debemos tener una manera de recuperar el orden original, de otro modo aunque asignemos valores de manera correcta no sabremos a que variables corresponden. Para resolverlo el programa asignamos dos campos: el $x_i.funcion_seleccion$ y $x_i.indice_original$; así ordenamos las variables según el criterio que necesitamos.

Los resultados se almacenan en otro archivo txt cuyo nombre lo podemos elegir o dejar vacío en cuyo caso será nombrado *Resultados*. Al ser sencillo este programa es fácil hacer algunas modificaciones, por ejemplo cambiar la función de selección. La línea de código que hace esto es la siguiente:

```
x[i].funcion_seleccion = x[i].beneficio / x[i].peso;  /* Esta línea */
asigna el valor de beneficio
por unidad de peso a la función de selección.
```

Que si cambiamos por alguna de las siguientes:

```
x[i].funcion_seleccion = x[i].beneficio; /* Esta únicamente toma */
el beneficio
x[i].funcion_seleccion = -x[i].peso;  /* Esta toma */
el peso
```

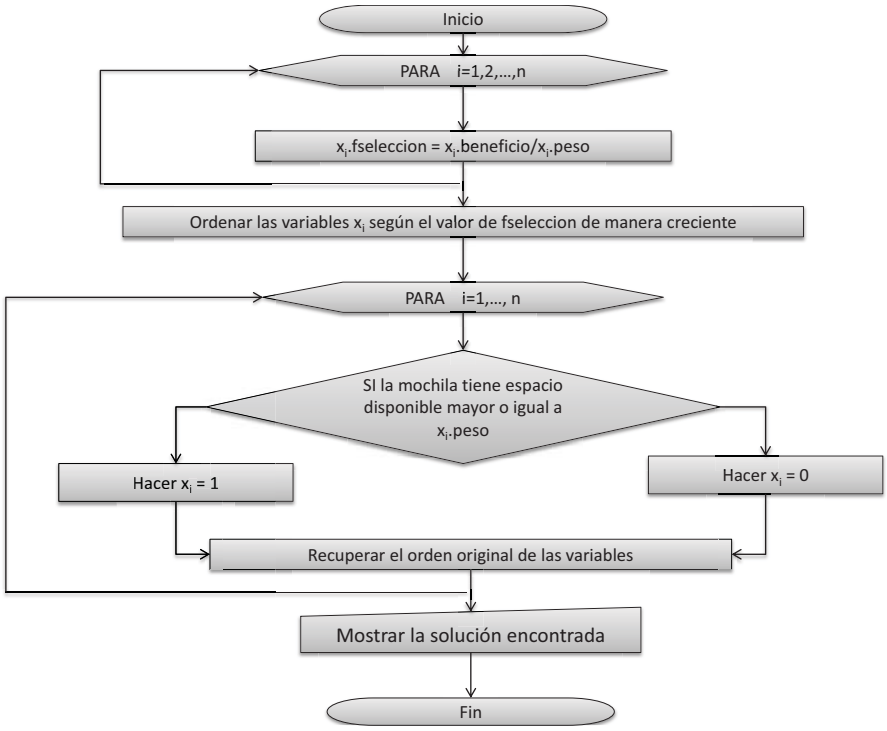

Figura 4.1: Diagrama del método glotón para el problema de la mochila

No es difícil deducir los cambios que ocasiona esto, la primera opción ordena los artículos según su beneficio y la segunda según su peso. Esta última tiene un signo menos porque los artículos son más atractivos si su peso es más pequeño.

Otra posibilidad de cambio es el tipo de variable, originalmente tenemos un problema binario, pero con un pequeño cambio podemos resolver la versión relajada del problema, debemos reemplazar el código:

```
if(ocupado + x[x.Count - i - 1].peso <= capacidad) /*Verificamos que
quede el espacio
suficiente para agregar el artículo */
{
x[x.Count - i - 1].valor = 1; /* En caso afirmativo el
artículo cabe y lo metemos a la mochila */
ocupado += x[x.Count - i - 1].valor * x[x.Count - i - 1].peso; /*Actua-
lizamos el espacio disponible */
}
```

Por la siguiente:

```
if (ocupado < capacidad) /* Únicamente revisamos que quede
```

```
espacio en la mochila */
{
x[x.Count - i - 1].valor =
Math.Min(1, (capacidad - ocupado) / x[x.Count - i - 1].peso );  /*En
caso de que no alcance para meter
todo el artículo metemos lo más que se pueda */
ocupado += x[x.Count - i - 1].valor * x[x.Count - i - 1].peso;
}
```

El primer y segundo renglón que están dentro del if deben ir en una misma línea. Cabe mencionar que las lineas que están entre los símbolos /* */ son comentarios dentro del programa.

4.2. Explicación del programa para el problema del agente viajero usando sistema de hormigas

Este programa es un tanto más complicado, para empezar la información es más, y debemos guardarla en más variables. El programa también funciona con archivos de texto, tanto para extraer la información como para escribir los resultados. El archivo con los datos de entrada debe llamarse *Datos_SHAV* y debe contener la información con el siguiente formato:

```
6
6
0.1
1
1
0.22
6
(1,2).10
(1,3).30
(1,4).4
(1,5).70
(1,6).44
(2,3).20
(2,4).80
(2,5).4
(2,6).50
(3,4).24
(3,5).60
(3,6).4
(4,5).13
(4,6).40
(5,6).11
```

Estos datos corresponden al ejemplo que dimos en el capítulo 2 en el siguiente orden: Número de hormigas, número de iteraciones, valor de tau, valor de alfa, valor de beta, valor de rho, número de nodos de la gráfica y finalmente las aristas de la gráfica y su distancia. Las aristas se representan como una pareja de números entre paréntesis y separados por comas, segudos de un punto y posteriormente el valor de la distancia.

La parte donde empieza la implementación del método es a partir del ciclo "while (t < num_iteraciones)" y podemos resumirlo en los siguientes pasos:

- Elegir los nodos donde iniciará cada hormiga
- Construir una solución por cada hormiga
- Actualizar el rastro de feromonas
- Actualizar el número de iteraciones y verificar si ya se han realizado el número necesario

De los cuales la construcción de soluciones es la más larga, cada una inicia en un nodo de los que determinamos en el paso anterior, posteriormente avanzamos hacia otros eligiendo cada uno con la probabilidad que definimos para este propósito hasta que se termine el ciclo hamiltoniano.

En la sección destinada a este método recomendamos establecer el número de hormigas como $|X|$ para colocar a una por nodo, sin embargo, como no es regla general el programa contempla una manera de asignarlos en caso de que sea un número distinto (digamos $n < |X|$).

Esta información la guardamos en una lista llamada nodos_iniciales de manera que el valor nodos_iniciales[i] almacena el nodo donde se iniciará la hormiga i, el resto del código lo explicamos en el diagrama 4.2.

El siguiente paso es construir una solución por cada hormiga; incluso en la sección dedicada a este tema mencionamos este proceso de manera separada para dejarlo más claro. Este proceso lo repetimos tantas veces como número de hormigas hayamos definido por lo que en el código lo repetimos tantas veces como elementos existan en la lista nodos_iniciales, eso se hace con la línea de código:

```
foreach(int j in nodos_iniciales)
```

que además nos sirve para posicionar a las hormigas en su nodo inicial. En esta parte almacenamos 3 tipos de información: el nodo donde está posicionada la hormiga al momento (nodo actual), el conjunto de nodos que ya han sido visitados (en el orden en que fueron recorridos) y un último conjunto que es el de nodos pendientes de visitar. Pensamos en un método que sólo resuelve instancias donde todos los nodos están conectados entre sí de modo todos los nodos de X están distribuidos en alguna de estas listas.

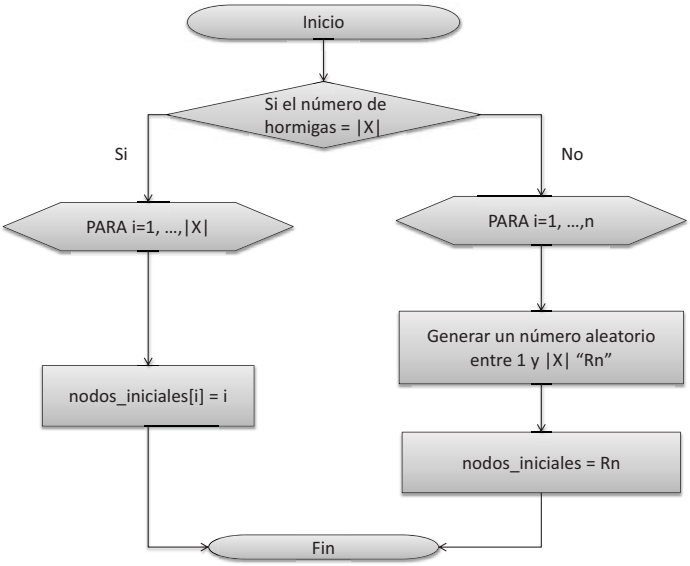

Figura 4.2: Generación del conjunto de nodos donde iniciarán las hormigas

Como ejemplo, supongamos que la primera hormiga inicia en el nodo 4, entonces este es el nodo actual, que añadiremos a los visitados y los posibles candidatos a visitar son todos los demás (si la gráfica tuviera 6 nodos serían 1,2,3,5 y 6). El diagrama 4.3 explica de una mejor manera este proceso.

También debemos mencionar que aquí se hace la comparación de la mejor solución encontrada con las que vayamos construyendo para que al final el programa pueda escribirla en un archivo txt.

La siguiente parte del código realiza la actualización del rastro de feromonas, misma que separamos en dos fases: evaporación de feromonas y actualización. La evaporación es sobre todos las aristas así que podemos hacerla recorriendo los elementos del conjunto A de la gráfica. En cambio, para el depósito de feromonas identificaremos los aristas aprovechando que para cada hormiga tenemos la lista de nodos visitados. El diagrama 4.4 resume el funcionamiento del programa.

De esta manera termina la iteración, lo que hace el programa después indicar que hemos hecho una iteración más (que es hacer $t = t + 1$) y verificar si ya se ha llegado al numero de iteraciones indicado.

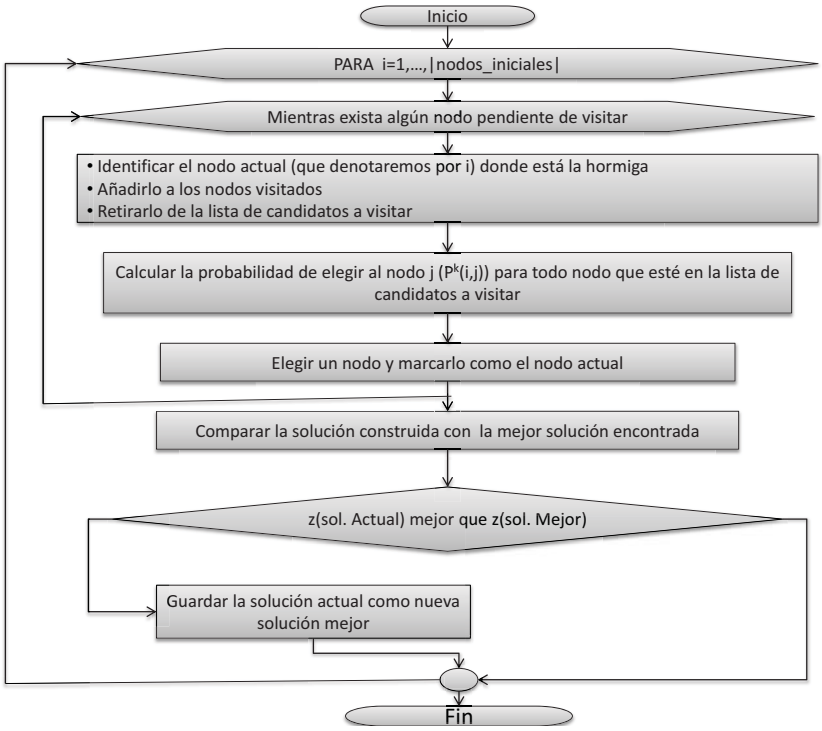

Figura 4.3: Construcción de las soluciones

4.3. Explicación del programa para el problema de número cromático usando recocido simulado

Recordemos que Recocido Simulado tiene 5 pasos principales:

- Construir una solución inicial
- Construir un vecino a partir de la solución actual
- Decidir si elegimos al vecino o no tanto para la siguiente iteración como para actualizar la mejor solución
- Actualizar la temperatura
- Verificar la condición de paro

El método no especifica la manera de construir una solución inicial, y comentamos que una alternativa es tomar la que se construye con un glotón, sin embargo, nosotros usamos como solución inicial la asignación de un color distinto a cada nodo. Esto porque era más fácil de programar ya que no hicimos

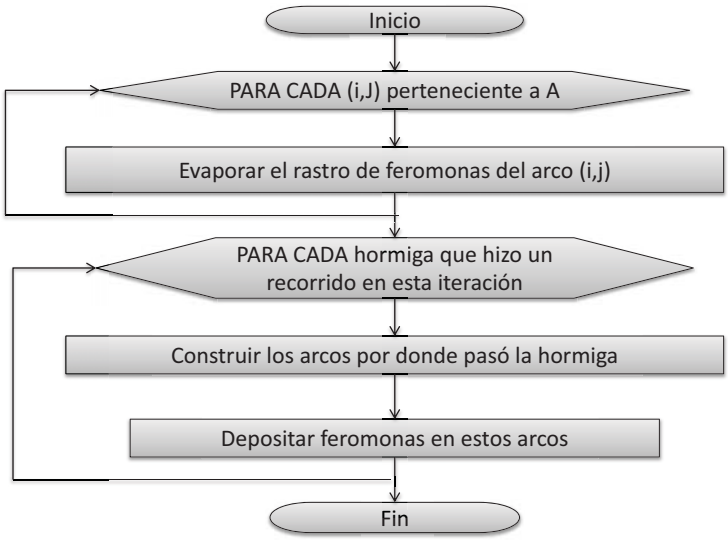

Figura 4.4: Actualización del rastro de feromonas

un programa para este problema usando el método glotón; hacer esto es fácil pues a cada nodo i le asignamos el color i.

El archivo de texto que contiene la información para el problema debe seguir el siguiente formato:

```
6
(1,4):(1,5):(1,6):(2,5):(3,5):(3,6)
4.2
0.7
4
5
```

El primer valor corresponde al número de nodos que tiene la gráfica, en la siguiente linea se representan las aristas con el mismo formato que manejamos antes, siguen el valor de tau, alfa, N y el número de iteraciones.

Una vez que hemos terminado la solución inicial debemos construir un vecino de la siguiente manera: elegimos de manera aleatoria un color para asignarlo a un nodo también elegido al azar, la única consideración especial que tuvimos en

esta parte fue para el caso donde el color elegido es el mismo que tiene el nodo en este momento, pues es el único caso donde no se genera un vecino. De modo que nuestra propuesta consiste en repetir el proceso hasta encontrar un vecino válido como se ilustra en el diagrama 4.5

La estrategia que empleamos era elegir sólo un vecino de manera aleatoria y no construir a todos por lo que basta con tener uno diferente para poder avanzar al siguiente paso que es comparar ambas soluciones. Recordemos que la función objetivo es:

$$z(s) = \sum_{j=1}^{k}(|C_j|)^2 - 2\sum_{j=1}^{k}|C_j||A_j|$$

Y si el valor de esta función es mayor elegimos esta última; pero en caso contrario debemos elegirla con cierta probabilidad. El resumen de este proceso aparece en la figura 4.6

En este momento no hemos hecho la comparación entre la mejor solución y la solución actual, no basta con reemplazarla si es mejor su función objetivo ya que manejamos soluciones no factibles. Por esto, en la sección del código que calcula la función objetivo metimos una variable binaria para revisar la factibilidad, si una solución tiene aristas malas implica que no es factible pues las aristas malas son aquellas entre nodos del mismo color, por otro lado si una solución no tiene aristas de este tipo sí es factible.

Las demás partes del código se pueden resumir rápidamente, pues sólo corresponden a actualizar la temperatura si es que se cumplen las condiciones y por último incrementar en una unidad el contador del número de iteraciones, así que lo presentaremos todo en un sólo diagrama más: el 4.7.

4.4. Manual de Usuario

Este programa sirve para resolver algunos problemas con métodos heurísticos que son el problema binario de la mochila por medio del método glotón, el problema de número cromático usando recocido simulado y los problemas del agente viajero y clique de cardinalidad máxima con sistema de hormigas.

Cada método fue programado por separado (el código de cada uno aparece en los anexos) de ahí que necesitemos los siguientes archivos ejecutables:

- Glotón Problema de la Mochila.exe
- RS para Num Cromatico.exe
- SH para el Clique Maximo.exe
- Sistema de Hormigas para el AV

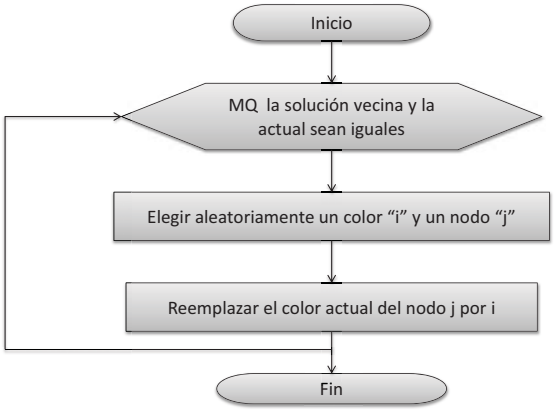

Figura 4.5: Construcción del vecino

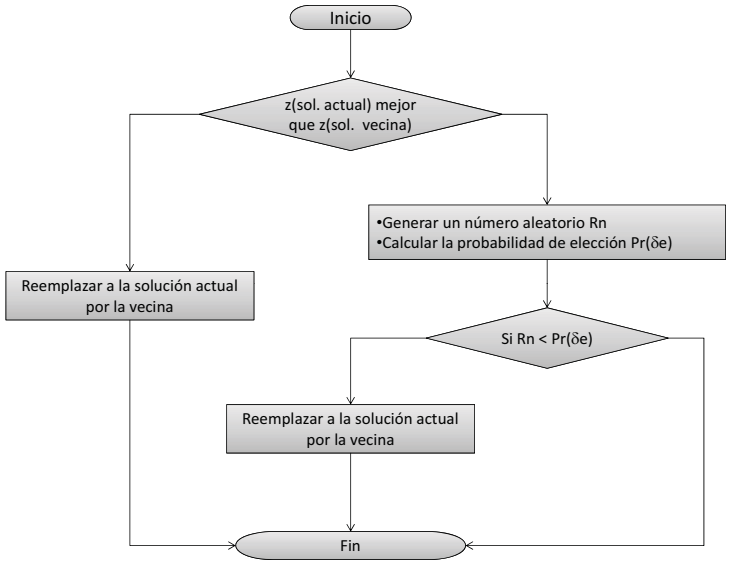

Figura 4.6: Comparación entre soluciones

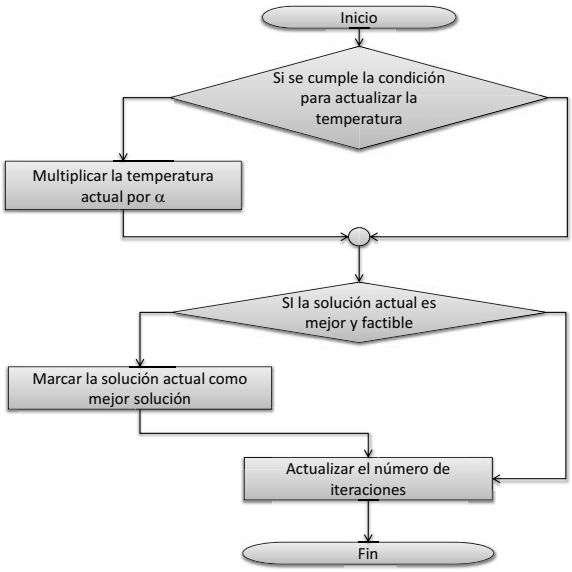

Figura 4.7: Actualización de la mejor solución, temperatura y número de iteraciones

En la sección *Explicación de los programas* detallamos el funcionamiento de estos programas, que extraen los datos del problema de un archivo de texto y escriben los resultados obtenidos en otro. El formato que tienen puede resultar poco amigable para el usuario por lo que agregamos otro programa llamado *Formulario.exe* por medio del cual el usuario podrá escribir los datos del problema y éste generará automáticamente el archivo de texto con el formato que necesitan los otros ejecutables.

Para garantizar el funcionamiento del programa hemos destinado esta sección a explicar un uso adecuado.

4.4.1. Requisitos

Como mencionamos se necesitan 5 archivos ejecutables, todos en una misma carpeta como se muestra en la figura 4.8 de los cuales solamente debemos abrir el archivo *Formulario.exe* y nos aparecerá una ventana con 3 pestañas, cada una sirve para meter los datos según el método elegido (glotón, sistema de hormigas o recocido simulado). Dentro de la pestaña dedicada a sistema de hormigas encontraremos otras dos pues recordemos que se pueden resolver dos problemas diferentes por este método, una imagen de esta ventana es la que se observa en la figura 4.9.

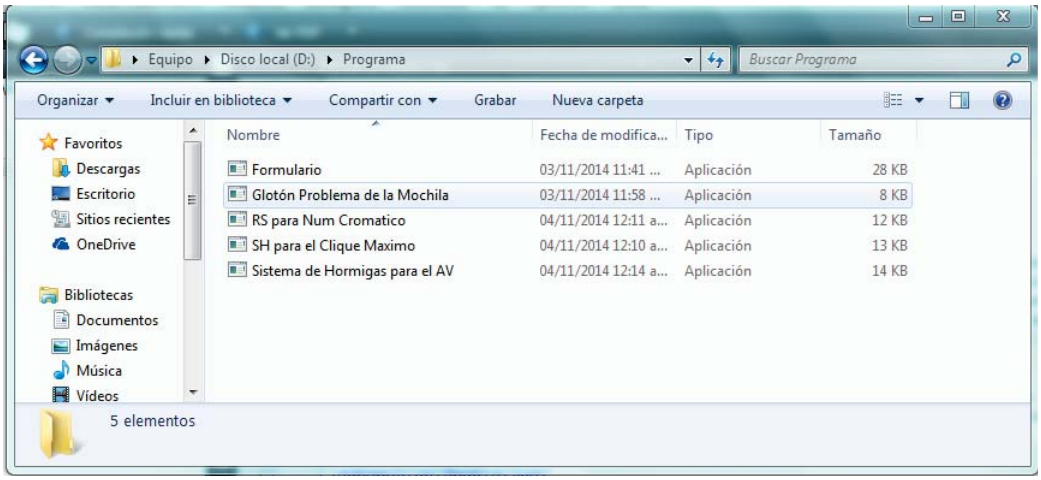

Figura 4.8: Todos los ejecutables van en la misma carpeta

4.4.2. Instrucciones de uso

Cada pestaña tiene un formulario donde se puede escribir la información de cada problema. Generalmente estos datos son números y se escriben en el formulario tal cual. No obstante hay 2 excepciones. En el problema de la mochila hay dos campos que son la primera excepción: *Pesos de los artículos* y *Beneficio de los artículos* en donde debemos añadir los beneficios y pesos de todos los artículos a la vez separados por comas; algo como la siguiente línea "2,3,4,5" representa los pesos (o beneficios) de 4 variables. La figura 4.10 muestra la manera correcta de rellenar el formulario para el problema de la mochila.

La segunda excepción es para los demás problemas que son de gráficas pues las aristas también se añaden juntos, escribiendolos entre paréntesis y separados por comas como la siguiente línea:

$$(1,2),(2,3),(3,4),(2,4),(4,1)$$

Una vez que se ha llenado el formulario por completo pasamos a presionar los dos botones que aparecen en cada pestaña. El botón "Guardar datos" generará el archivo de texto que requiere cada uno de los demás ejecutables y posteriormente debemos seleccionar el botón "Resolver" que ejecutará el programa correspondiente para resolver el problema.

Al seleccionar el segundo botón se abrirá una ventana (como la figura 4.11) que nos muestra la información que hemos registrado con el fin de validarla, en caso de ser correcta seleccionamos la opción Aceptar; en caso contrario seleccionamos cancelar y modificamos la información del formulario (es importante volver a seleccionar el botón de guardar para poder actualizar el archivo de texto).

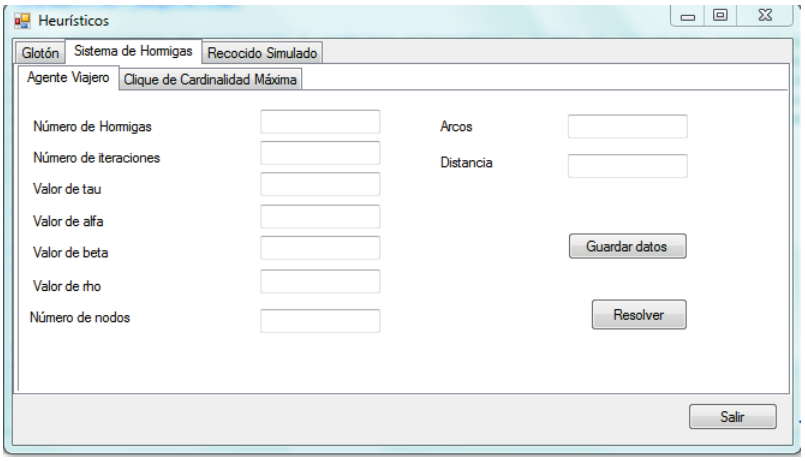

Figura 4.9: Ventana que aparece al abrir *Formulario.exe*

Finalmente, si seleccionamos la opción aceptar se abrirá una ventana de símbolos de sistema que nos confirmará que el problema ha sido resuelto y el nombre del archivo donde se ha guardado la información, ese archivo se encuentra en la misma carpeta donde se guardaron los ejecutables.

Si los datos que se han metido no cumplen con los requisitos del método (como que en el de la mochila sean 5 pesos y 4 beneficios) o no son compatibles con el programa (como meter letras en el formulario o no escribir la información de las aristas con el formato que explicamos) el programa mostrará en la ventana que ha ocurrido un error y el problema no se pudo resolver.

Para cerrar la ventana de símbolos de sistema presionamos cualquier tecla; para cerrar el formulario debemos seleccionar el botón “Salir” que borrará los archivos de texto que guardaban la información del problema para evitar errores en usos posteriores del formulario.

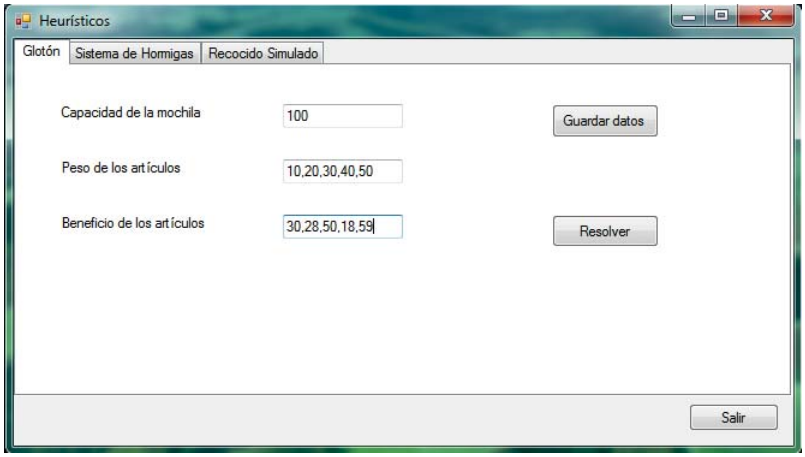

Figura 4.10: Formulario ya completado

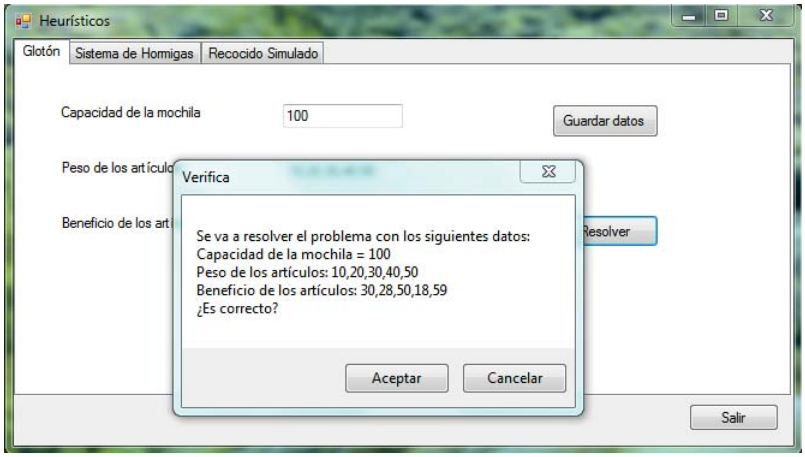

Figura 4.11: Ventana de confirmación

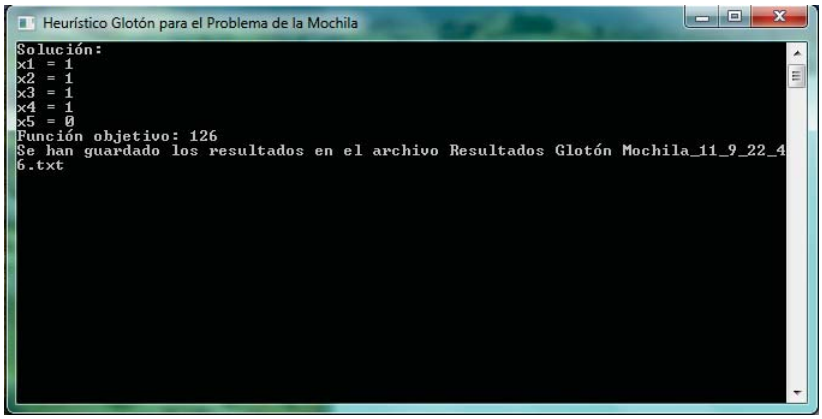

Figura 4.12: Ventana de sistema que aparece en un caso exitoso

Conclusiones

Resolver un problema difícil es en sí un problema, ya que el uso de algoritmos que encuentran el óptimo implican un costo muy alto con respecto al tiempo, aún cuando se haga uso de una computadora por lo que usar un heuristico vale la pena. El tiempo es su principal ventaja ya que para poder dar una solución a tiempo es mucho mejor que presentar una con retraso aún cuando ésta sea una aproximación. No obstante se tienen más ventajas con el uso de heurísticos, como la facilidad de programar ya que son instrucciones mas breves y fáciles de entender por su lógica.

Por otro lado, tenemos la ventaja de que estos métodos son más personalizables, como mencionábamos en el capítulo 1 las instrucciones de los métodos son más generales, como en el caso del método de mejoras sucesivas donde se busca al mejor vecino de las vecindad, sin especificar quién es el vecino y cuál es la vecindad permitiéndonos definir cada elemento según nos convenga, esto a su vez permite que en caso de detectar una falla en el método sea posible modificarlo para aumentar su eficiencia. Finalmente es importante mencionar que en caso de tener duda con la eficiencia del método es fácil apoyarse en otro método así como en repetir el mismo algoritmo cambiando los parámetros.

Sin embargo, debemos tener en cuenta que estas mismas características implican un riesgo, pues debemos elegir bien el método para que se obtenga un desempeño aceptable y mejorar la calidad del método no solo es una sugerencia, es un requisito para conseguir un buen resultado ya que hemos mostrado casos donde éstos métodos encuentran una mala solución.

También cabe mencionar que éstos métodos muestran su mayor cualidad cuando tratamos problemas grandes por lo que la rapidez puede no ser tan evidente en instancias pequeñas como las que mostramos en los ejemplos, pero debe notarse que éstos fueron para explicar el funcionamiento y aplicación principalmente. Para probarlos en instancias grandes es conveniente hacer uso del programa que se diseñó y mencionó en el capítulo 4.

Bibliografía

[1] Alonso S., Conrdon O., and Fernandez I. y Herrera F. *La Metaheurística de Optimización Basada en Colonias de Hormigas: Modelos y Nuevos Enfoques*.

[2] Glover, F. and Melián, B. . Tabu search. 2003.

[3] Hernández, M.C. . *Introducción a la teoría de redes*. Aportaciones matemáticas: textos. Sociedad Matemática Mexicana, 1997.

[4] Martí, Rafael. *Algoritmos Heurísticos en Optimización Combinatoria*. Universidtat de València. Facultat de Matem'atiques. Departament d' Estadística i Investigació Operativa.

[5] Martí, Rafael. *Procedimientos Metaheurísticos en Optimización combinatoria*. Universidtat de València. Facultat de Matem'atiques. Departament d' Estadística i Investigació Operativa.

[6] Martín, Elena. Sistemas heurísticos y selección de indicadores, 2003.

[7] Sakarovitch, M. *Optimisation combinatoire: Programmation discrète*. Hermann, 1984.

[8] Downsland, K. y Adenzo, B. Heuristic design and fundamentals of the simulated annealing. 2003.

[9] Brassard, G.A. y Bratley, P.A. *Fundamentos de Algoritmia*. Fuera de colección Out of series. Prentice-Hall International, 1997.

[10] Lee, K.Y. y El-Sharkawi, M.A. *Modern Heuristic Optimization Techniques: Theory and Applications to Power Systems*. IEEE Press Series on Power Engineering. Wiley, 2008.

[11] Ponce, J. Ponce, E. Padilla, A. Padilla, F. y Ocoha A. *Algoritmo de Colonia de Hormigas para el Problema del Clique Máximo con un Optimizador Local K-opt*. 2006.

[12] Taha, H.A. y Pozo, V.G. *Investigación de operaciones*. Pearson Educación, 2012.

[13] Parker, R.G. y Rardin, R.L. *Discrete optimization*. Computer science and scientific computing. Academic Press, 1988.

[14] Papadimitriou, C.H. y Steiglitz, K. *Combinatorial Optimization: Algorithms and Complexity*. Dover Books on Computer Science Series. Dover Publications, 1998.

ANEXOS

Algunos problemas NP Completos

En 1979 Graey y Jhonson publicaron un libro con la recopilación de 300 problemas pertenecientes a la clase NP-C y actualmente son más los que se han demostrado que pertenecen a la clase, todo esto debido a la demostración de Cook sin embargo, estas demostraciones no fueron directas del problema SAT, pues este no era muy conocido dentro de la optimización discreta.

La persona que hizo una relación entre el problema SAT y algunos otros más conocidos en este campo fue Richard Karp quien, en 1972, realizó una lista de 20 problemas pertenecientes a la clase NP-C a partir de la demostración de Cook.

La lista de estos problemas y el problema que Karp redujo polinomialmente para su demostración es la que se presenta a continuación.

Lista de problemas NP Completos:

- El problema SAT
  - El problema de programación lineal entero binario
  - El problema del clique de cardinalidad máxima
    - El problema de empacamiento de conjuntos (Set packing)
    - El problema de cubiertas de vértices (Vertex cover)
      - El problema de cubiertas (Set covering)
      - El problema Feedback vertex set
      - El problema Feedback arc set
      - El problema del ciclo Hamiltoniano dirigido
        - El problema del ciclo Hamiltoniano no dirigido
  - El problema 3-SAT (Problema SAT con 3 variables por cláusula)
    - El problema del número cromático
      - El problema de cubierta de cliques
      - El problema de la cubierta exacta (Exact cover)
        - El problema Hitting Set
        - EL problema del árbol de Steiner
        - El problema de 3-Acoplamiento
        - El problema binario de la mochila
          - El problema de secuencia de trabajos
          - El problema de la partición
            - El problema del corte máximo

A continuación presentaremos algunas de estas demostraciones que serán con el método que describimos en la sección anterior: demostraremos que el problema pertenece a la clase NP y posteriormente reduciremos un problema NP-C conocido a éste. Adicionalmente haremos una descripción del problema y del problema de optimización que está asociado al problema.

Problema 3-SAT

El problema 3-SAT es un caso particular del problema SAT con la condición adicional de que cada cláusula debe tener exactamente 3 variables (es decir, cada $C_i = \{u_{i1} \vee u_{i2} \vee u_{i3}\}$).

Teorema 4.1 *El problema 3-SAT es NP Completo.*

Demostración. Para demostrar que el problema pertenece a la clase NP recordemos el concepto del supervisor, es decir, que dada una combinación de variables de X para las que $E = 1$ debemos poder convencer a otra persona en un tiempo polinomial que la respuesta es verdadera.

Si tenemos la combinación de variables para la cual el enunciado es verdadero y nuestro problema tiene m cláusulas entonces tendríamos que hacer m^3 operaciones elementales para comprobar que $E = 1$ es decir, lo haríamos en un tiempo polinomial lo que implica que el problema 3-SAT está en la clase NP.

Para demostrar que está en NP-C reduciremos el problema SAT a un problema 3-SAT para el cual se pueden presentar 3 opciones por cada cláusula dependiendo de la instancia:

- La cláusula C_i tiene exactamente 3 variables.
- La cláusula C_i tiene menos de 3 variables.
- La cláusula C_i tiene más de 3 variables.

Cuando se nos presenta el primer caso es evidente que no hay que realizar ninguna acción, en el caso de que se presente la segunda opción la acción es sencilla pues basta con repetir una de las variables del enunciado tanto como sea necesario por ejemplo, si tenemos la cláusula:

$$C_i = \{x_1 \vee x_2\}$$

Tenemos la posibilidad de cambiarla por la cláusula:

$$C_i = \{x_1 \vee x_2 \vee x_1\} \text{ o también } C_i = \{x_1 \vee x_2 \vee x_2\}$$

Si la cláusula tiene más de tres variables, es decir $C_i = \{u_{i1}, u_{i2}, ..., u_{ik}\}$ con $k > 3$ dividiremos la cláusula como sigue:

$$\begin{aligned} C_{i1} &= \{u_{i1} \vee u_{i2} \vee \lambda_1\} \\ C_{i2} &= \{\overline{\lambda}_1 \vee u_{i3} \vee \lambda_2\} \\ &\vdots \\ C_{i,k-2} &= \{\overline{\lambda}_{k-3} \vee u_{ik-1} \vee u_{ik}\} \end{aligned}$$

La idea es que la cláusula C_i del problema original sea equivalente a la combinación de cláusulas:

$$C_{i1} \wedge C_{i2} \wedge ... \wedge C_{ik-2}$$

La explicación del porqué son equivalentes es la siguiente:

La cláusula C_i es verdadera si al menos un $u_{ij} = 1$ por otro lado nuestra sustitución es verdadera si cada $C_{ij} = 1$. Notemos que si $\lambda_{ij} = 1$ tenemos que $k-1$ de las cláusulas serían verdaderas excepto la cláusula $C_{i,k-2}$ la cual sería igual a 1 si $u_{i,k-1}$ ó u_{ik} son verdaderas y de la misma manera es posible encontrar una combinación de las λ_{ij} para las que $k-1$ cláusulas sean verdaderas y la otra cláusula es igual a 1 si $u_{ij} = 1$ y todas son falsas si todas las $u_{ij} = 0$.

De esta manera reduciremos polinomialmente nuestro problema. Por lo tanto el problema 3-SAT es NP Completo. ■

Problema del Clique de Cardinalidad Máxima

Dada una gráfica $G = (X, A)$ donde X es el conjunto de nodos y A el de aristas un *clique* C es un conjunto de nodos tales que cada nodo de C es adyacente a los demás nodos de C. En la figura 4.13 tenemos una gráfica en la que los nodos 1,2 y 3 forman un clique y los nodos 1,2 4 no ya que los nodos 2 y 4 no son adyacentes.

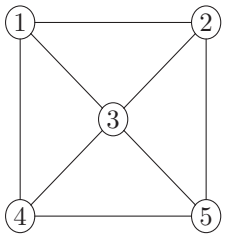

Figura 4.13: Ejemplo de un clique formado por los nodos 1,2 y 3

Al problema de encontrar el clique con cardinalidad más alta en la gráfica lo conocemos como *Problema del clique de cardinalidad máxima.*

El problema del clique de cardinalidad máxima es el siguiente: Dada la gráfica $G = (X, A)$ y un entero positivo k ¿Existe un clique C tal que $|C| \leq k$?

Teorema 4.2 *El problema del clique es NP Completo*

Demostración. Este problema está en NP debido a que un clique es un conjunto y para verificar la cardinalidad de un conjunto de tamaño n basta con contar los elementos del conjunto, es decir, se requieren n operaciones elementales y dada la cardinalidad de un conjunto comparar ese número con un entero positivo k se hace con una operación elemental por lo tanto, el número de operaciones elementales necesarias para verificar que dado un clique su cardinalidad es menor

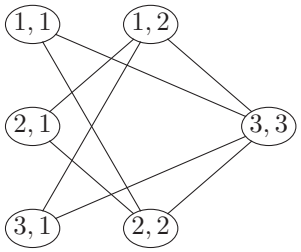

Figura 4.14: Transformación de las cláusulas del problema SAT a una gráfica

o igual a un número k está acotado por una función $O(n)$.

La demostración de que el problema está en NP se hace mediante una reducción polinomial del problema SAT.

Dado un problema SAT con un conjunto de variables $X = \{x_1, x_2, ..., x_n\}$ y m cláusulas C_i construiremos la gráfica $G = (X', A')$ de la siguiente manera:

$X' = \{[i, j] |$ La variable x_i ó $\overline{x}_i$ es una variable de la cláusula $C_j\}$
$A' = \{([a, i], [b, j]) | i \neq j$ y $a \neq b\}$

La gráfica funciona ya que no existen aristas que unan a la variable x_i con $\overline{x}_i$ por lo que no se le asignará el valor de 1 a una variable y a su complemento. Además la gráfica asigna los valores de cada x_i basándose en todas las cláusulas al no haber aristas entre las variables ubicadas en la misma cláusula.

Con esta gráfica buscaremos un clique de cardinalidad k; el conjunto de nodos que pertenezcan al clique serán las variables que tengan valor de 1. Por lo tanto, el problema del clique es NP Completo ■

Para ejemplificar esta transformación tomaremos nuevamente el problema SAT.

$$E = (x_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee x_2) \wedge (x_3).$$

Lo cual genera la gráfica de la figura 4.14.

En la que los únicos cliques que aparecen corresponden a las soluciones $x_1 = x_2 = x_3 = 1$ y $x_1 = 0, x_2 = x_3 = 1$ que como habíamos mencionado son las combinaciones para las cuales la cláusula es verdadera.

Número cromático

Dada una gráfica $G = (X, A)$ y un entero positivo $k < |A|$, ¿Es posible colorear los nodos de X con k o menos colores de manera que ninguna pareja de nodos adyacentes tengan el mismo color?

El problema de optimización consiste en buscar el mínimo número de colores para colorear la gráfica G.

Teorema 4.3 *El problema del número cromático es NP Completo.*

Demostración. La demostración se hará a través de la reducción polinomial de un problema 3-SAT a uno de número cromático, pero primero observemos que para verificar que una gráfica coloreada con un número s de colores no tiene nodos adyacentes de un mismo color se puede hacer con un número de comparaciones igual a la cardinalidad de A y validar que este número s es menor o igual a k requiere una operación elemental por lo que el número de operaciones está acotado por una función $O(n)$ por lo que el problema está en NP.

Para la reducción polinomial haremos una reducción de un problema 3-SAT a uno de número cromático con $k \leq 3$ y se hace construyendo una gráfica compuesta por 3 tipos de subgráficas. Primero tendemos 3 nodos que llamaremos V,F y Z. Estos 3 serán adyacentes entre sí como se aprecia en la figura 4.15:

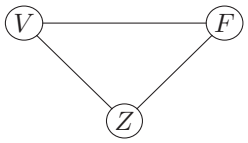

Figura 4.15: Primera parte de la gráfica

Estos nodos tendrán 3 colores ya que son adyacentes entre sí y nos ayudarán a definir los siguientes colores para nuestra gráfica. El segundo tipo de subgráficas lo tendremos formado por nodos que representen a las literales x_i y $\overline{x}_i$ los cuales serán adyacentes entre sí y adyacentes al nodo Z. De esta manera garantizamos que estos nodos sean coloreados con el mismo color que el nodo V o el nodo F y siempre sucederá que las literales y sus complementos serán de distintos colores por ser adyacentes entre sí (Figura 4.16).

Y el tercer tipo de subgráfica (figura 4.17) será el que nos ayude a colorear a G. está compuesto por 5 nodos y 10 aristas y tendremos una de estas subgráficas por cada cláusula del problema; 3 nodos de esta subgráfica serán adyacentes a los nodos de las literales que aparecen en la cláusula y los otros 2 nodos serán adyacentes al nodo V. Ésta será de la siguiente manera:

La combinación de literales para los que $E = 1$ serán obtenidas a través de la coloración pues las literales que tengan el mismo color que el nodo V serán las

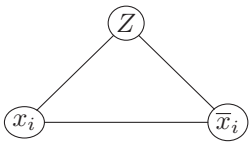

Figura 4.16: Segunda parte

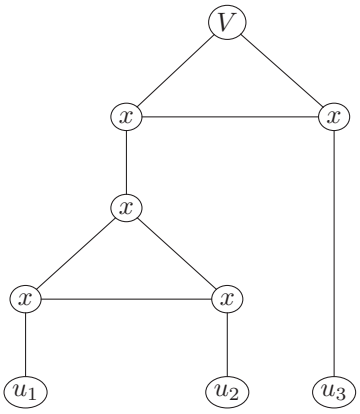

Figura 4.17: Tercera parte de la gráfica

verdaderas y las falsas aquellas que sean coloreadas con el color F, sabemos que todas las literales serán coloreadas con estos dos colores porque son adyacentes al nodo X y al ser adyacentes entre sí no tendrán el mismo color. La única manera en que alguna literal no sea coloreada del mismo color que V o F será si necesitamos más de 3 colores para colorear la gráfica y en ese caso la respuesta al problema de coloración será no y por consecuencia la del problema 3-SAT también.

Cubierta Exacta

Dada una familia $S = \{S_i\}$ ¿Existe una subfamilia $W \subseteq S$ tal que los $W_i \in W$ sean ajenos y $\cup W_i = \cup S_i$

Teorema 4.4 *El problema de cubierta exacta es NP Completo.*

Demostración. La demostración la haremos reduciendo el problema de número cromático a un problema de cubierta exacta. Para probar que el problema es NP definimos el tamaño del problema como la cardinalidad de $\cup S_i$ que podemos llamar n; observemos que verificar que los conjuntos $\cup W_i$ y $\cup S_i$ son iguales podemos ordenarlos y comparar entrada a entrada los elementos de cada conjunto ya que hemos visto que existen algoritmos de ordenamiento que son polinomiales. Lo más tardado es verificar que los conjuntos de W son ajenos, para ello podemos tomar el conjunto $\tilde{W} = \cup W_i$ en el que buscaremos cada elemento $w \in W_i$ de cada conjunto de la subfamilia W y cada que encontremos a un w lo descartaremos de $\tilde{W}$ de esta manera en la primera iteración a lo más haremos n comparaciones, en la segunda $n-1$ y así sucesivamente hasta llegar a 1 por lo que el total de iteraciones a lo más será de $\dfrac{n(n+1)}{2}$ que también es una función polinomial.

La idea de la reducción consiste en representar a los S_j como subconjuntos de X lo que implicará que la unión de los S_j nos dará a todo el conjunto y al ser ajenos cada conjunto puede ser coloreado con el mismo color, siempre y cuando los nodos no sean adyacentes lo cual se obtiene de la siguiente manera. Dada la gráfica $G = (X, A)$ y un entero k definimos los conjuntos de la siguiente manera:

$$S = X$$
$$S_j = \{i : i \text{ es adyacente a } j \text{ en } G\}$$

Por lo tanto, el problema es NP Completo. ■

Tomemos nuevamente a la gráfica bipartita para ejemplificar esta reducción, sabemos que nuestro problema de número cromático es verdadero para $k \leq 2$. Aplicando la reducción tenemos 6 conjuntos S_j que son:

$$S_1 = S_2 = S_3 = \{4, 5, 6\} \quad \text{y}$$
$$S_4 = S_5 = S_6 = \{1, 2, 3\}$$

Es claro que con dos conjuntos de estos (por decir un ejemplo tomemos a S_1 y S_4) obtenemos una cubierta exacta, por lo que el problema de decisión es verdadero para $k \leq 2$. Y aunque hay varias soluciones a nuestro problema de cubierta exacta notemos que todas recuperan la misma solución: El conjunto de nodos 1,2 y 3 son de un color y el 4,5 y 6 son de otro.

Problema binario de la Mochila

El problema de la mochila es un problema de optimización en el que se busca maximizar una función objetivo sin que se sobrepase de cierta capacidad, es decir, se debe cumplir la restricción $\sum a_j x_j \leq b$ y si problema de decisión asociado es el siguiente: dados los enteros $\{a_1, a_s 2, ..., a_n\}$ y b. ¿Existe un vector x binario tal que $\sum a_j x_j = b$?

Teorema 4.5 *El problema binario de la mochila es NP Completo.*

Demostración. Para probar que el problema es NP basta con verificar que el número de operaciones elementales necesarias para verificar que el vector x cumple que $\sum a_j x_j = b$ es igual a n+1 pues se necesitan $2n+1$ multiplicaciones para calcular los productos $a_j x_j$ y n sumas para obtener $\sum a_j x_j$ y una operación para verificar que esta suma es igual a b por lo tanto la función es $O(n)$.

Con respecto a la reducción polinomial la demostración se hará reduciendo el problema de cubierta exacta, pero esta vez a uno de la mochila. Sea:

$$d = |S| + 1$$
$$a_{ij} = \begin{cases} 1 & \text{Si } u_i \in S_j \text{i} = 1,2,...,|S| \\ 0 & \text{En otro caso.} \end{cases}$$
$$a_j = \sum_i a_{ij} d^{i-1}$$
$$b = \sum_i d^{i-1}$$

Observemos que hay tantos elementos a_j como conjuntos S_j lo que indica que las variables para las cuales a_j sea igual a 1 indicarán que S_j formará parte de la cubierta. El valor d^{i-1} es lo que garantiza que los u_i no se repitan pues tienen un peso diferente cada uno, pero es el mismo peso en cada restricción por lo que no hay forma de que se elijan conjuntos S_j que sean ajenos pero que no cubran al conjunto. La reducción nos permite afirmar que el problema es NP Completo. ■

Secuencia de Tareas

Dado un conjunto de trabajos $\{1, 2, ..., t\}$ con tiempos de realización $\{\tau_1, \tau_2, ..., \tau_t\}$ enteros, un conjunto de fechas (también entero) $\{d_1, d_2, ..., d_t\}$ y un vector de penalización $\{P_1, P_2, ..., P_t\}$ y un entero positivo k, existe un ordenamiento de los trabajos $\pi = (\pi_1, \pi_2, ..., \pi_t)$ tal que el tiempo de penalización no sea mayor a k. Notemos que para el trabajo j se tiene una penalización P_{π_j} si $\tau_{\pi_1} + \tau_{\pi_2} + ... + \tau_{\pi_j} \geq d_{\pi_j}$ (y cero en otro caso). En la versión de optimización queremos que la secuencia de trabajos sea tal que el tiempo de penalización sea mínimo.

Teorema 4.6 *El problema de secuencia de trabajos es NP Completo.*

Demostración. La demostración se hará reduciendo el problema de la mochila a uno de secuencias de la siguiente manera. Y para verificar que el problema es NP observemos dada una secuencia π de t tareas las penalizaciones se calcularán con $\frac{t(t+1)}{2} + t$ operaciones ya que la primera penalización se calcula comparando π_1 con d_{π_1} (1 operación), la segunda comparando $\pi_1 + \pi_2$ con d_{π_2} (1 operación por la suma y 1 operación por la comparación) y así hasta llegar a π_t que compara a $\pi_1 + \pi_2 + ... + \pi_t$ con d_{π_t} (t operaciones por la suma y 1 operación por la comparación). Esta función es una función $O(t^2)$ que es polinomial. Para la reducción polinomial sean:

$$t = r,\ \tau_i = P_i = a_i\ \forall i \text{ y } d_i = b$$

Es decir, tenemos tantos trabajos como variables a_i, el tiempo de producción es igual al peso de cada artículo del problema de la mochila (los a_i) al igual que la penalización y la fecha límite para realizar las tareas es la misma para todos b. Por lo tanto se deben realizar todas las tareas antes de b, claro está que si la suma de los a_i es mayor a b esto no sería posible aunque nuestro problema de la mochila si tenga solución, por lo que debemos hacer $k = \sum_i a_i - b$. ■

Código para resolver un problema de la Mochila usando un Método Glotón

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IO;

namespace Glotón_Problema_de_la_Mochila
{
    class Candidato
    {
        public int indice_original = new int();
        public double valor = new double();
        public double peso = new double();
        public double beneficio = new double();
        public double funcion_seleccion = new double();
    }
    class Program
    {
        public double Ordena(double x, double y)
        {
            return x.CompareTo(y);
        }
        static void Main(string[] args)
        {
            Console.Title = "Heurístico Glotón para el Problema de la
Mochila";
            List<Candidato> x = new List<Candidato>();
            double capacidad = new double();
            StreamReader sr = new StreamReader("C:\\Datos.txt");
            string linea = sr.ReadLine();
            linea = sr.ReadLine();
            capacidad = Double.Parse(linea);
            linea = sr.ReadLine();
            linea = sr.ReadLine();
            string[] separados = linea.Split(',');
            Candidato vacio = new Candidato();
            for (int i = 0; i < separados.Length;i++ )
            {
                vacio.peso = Double.Parse(separados[i]);
                x.Add(vacio);
                vacio = new Candidato();
            }
            linea = sr.ReadLine();
            linea = sr.ReadLine();
            separados = linea.Split(',');
            for (int i = 0; i < separados.Length;i++ )
            {
                x[i].beneficio = Double.Parse(separados[i]);
                x[i].indice_original = i + 1;
            }
            sr.Close();
            for (int i = 0; i < x.Count;i++ )
```

```
            {
                x[i].funcion_seleccion = x[i].beneficio / x[i].peso;
                // x[i].funcion_seleccion = x[i].beneficio;
                // x[i].funcion_seleccion = -x[i].peso;
            }
            x = x.OrderBy(p => p.funcion_seleccion).ToList();
            double ocupado = 0;
            for (int i = 0; i < x.Count;i++)
            {
                if(ocupado + x[x.Count - i - 1].peso <= capacidad)
                {
                    x[x.Count - i - 1].valor = 1;
                    ocupado += x[x.Count - i - 1].valor * x[x.Count -
i - 1].peso;
                }
                else
                {
                    x[x.Count - i - 1].valor = 0;
                }
            }
            x = x.OrderBy(p => p.indice_original).ToList();
            double f_objetivo = new double();
            for (int i = 0; i < x.Count; i++)
            {
                f_objetivo += x[i].valor * x[i].beneficio;
            }
            Console.WriteLine("Introducir el nombre del archivo donde
se guardarán los resultados");
            string nombre = Console.ReadLine();
            if(nombre == "")
            {
                nombre = "Resultado";
            }
            StreamWriter sw = new StreamWriter("C:\\"+nombre+".txt");
            sw.WriteLine("Resultado");
            for (int i = 0; i < x.Count; i++)
            {
                sw.WriteLine("x{0} = {1}",i+1,x[i].valor);
            }
            sw.WriteLine("Función objetivo: {0}",f_objetivo);
            sw.Close();
            Console.WriteLine("Se han guardado los resultados en el
archivo \""+ nombre +".txt\" ");
            Console.ReadKey();
        }
    }
}
```

Código para resolver un problema del Agente Viajero usando Sistema de Hormigas

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IO;

namespace SH_para_el_Agente_Viajero
{
    public class Hormiga
    {
        public int nodo_actual = new int();
        public List<int> Visitados = new List<int>();
        public List<int> Candidatos = new List<int>();
        public double Distancia = 0;

        public List<int> ObtenerCandidatos(Hashtable A, List<int> X,
StreamWriter sw)
        {
            sw.WriteLine("Se tienen disponibles los siguientes nodos
para recorrer:");
            List<int> candidatos = new List<int>();
            for (int i = 0; i < Visitados.Count; i++)
            {
                X.Remove(Visitados[i]);
            }
            for (int i = 0; i < X.Count; i++)
            {
                if (A.ContainsKey("(" + nodo_actual + "," + X[i] +
")") || A.ContainsKey("(" + X[i] + "," + nodo_actual + ")"))
                {
                    candidatos.Add(X[i]);
                    sw.Write(X[i] + "    ");
                }
            }
            sw.WriteLine();
            return candidatos;
        }
        public Hashtable ObtenerProbabilidades(Hashtable A, double
alpha, double beta, StreamWriter sw)
        {
            sw.WriteLine("Cada uno con la siguiente probabilidad:");
            Hashtable Probabilidades = new Hashtable();
            List<double> probas = new List<double>();
            double[] datos = new double[3];
            double proba;
            double suma = 0;
            for (int i = 0; i < Candidatos.Count; i++)
            {
                if (A.ContainsKey("(" + nodo_actual + "," +
Candidatos[i] + ")"))
                {
                    datos = (double[])A["(" + nodo_actual + "," +
Candidatos[i] + ")"];
```

```
                }
                else
                {
                    datos = (double[])A["(" + Candidatos[i] + "," +
nodo_actual + ")"];
                }
                proba = Math.Pow(datos[1], alpha) * Math.Pow(datos[2],
beta);
                probas.Add(proba);
                suma += proba;
            }
            for (int i = Candidatos.Count - 1; i > -1; i--)
            {
                Probabilidades.Add(Candidatos[i], probas[i] / suma);
                sw.WriteLine("El nodo {0} se elige con probabilidad:
{1}", Candidatos[i], probas[i] / suma);
            }
            return Probabilidades;
        }
        public int NodoSiguiente(Hashtable Probabilidades,
StreamWriter sw, Random aleatorio)
        {
            int j = 0;
            double m = aleatorio.NextDouble();
            double acumulado = 0;
            foreach (DictionaryEntry nb in Probabilidades)
            {
                if (acumulado < m && m <= acumulado +
(double)nb.Value)
                {
                    j = (int)nb.Key;
                }
                acumulado += (double)nb.Value;
            }
            sw.WriteLine("De donde se elige el nodo {0} porque el
número aleatorio generado fue: {1}", j, m);
            m = 0;
            return j;
        }
        public double ObtenerDistancia(Gráfica G)
        {
            double f_objetivo = 0;
            List<string> arcos = new List<string>();
            string arco;
            for (int i = 0; i < Visitados.Count - 1; i++)
            {
                if (G.A.ContainsKey("(" + Visitados[i] + "," +
Visitados[i + 1] + ")"))
                {
                    arco = "(" + Visitados[i] + "," + Visitados[i + 1]
+ ")";
                }
                else
                {
                    arco = "(" + Visitados[i + 1] + "," + Visitados[i]
+ ")";
                }
                arcos.Add(arco);
```

```
            }
            double[] matriz = new double[3];
            for (int i = 0; i < arcos.Count; i++)
            {
                matriz = (double[])G.A[arcos[i]];
                f_objetivo += matriz[0];
            }
            return f_objetivo;
        }
        public void SolucionObtenida(Gráfica G, StreamWriter sw)
        {
            for (int i = 0; i < Visitados.Count; i++)
            {
                sw.Write(Visitados[i]);
            }
            sw.WriteLine("");
            sw.WriteLine("Con una distancia total recorrida de: {0}",
Distancia);
        }
    }

    public class Gráfica
    {
        public List<int> X = new List<int>();
        public Hashtable A = new Hashtable();
        public void MuestraInfo()
        {
            double[] matriz = new double[3];
            foreach (DictionaryEntry nb in A)
            {
                matriz = (double[])nb.Value;
                Console.WriteLine("Arco: {0}  Tau: {1}  Eta: {2}
Distancia: {3}", nb.Key, matriz[1], matriz[2], matriz[0]);
            }
        }
        public void EvaporaRastro(double rho)
        {
            List<string> arcos = new List<string>();
            string arco;
            foreach (DictionaryEntry nb in A)
            {
                arco = (String)nb.Key;
                arcos.Add(arco);
            }
            double[] matriz = new double[3];
            for (int i = 0; i < A.Count; i++)
            {
                matriz = (double[])A[arcos[i]];
                matriz[1] *= (1 - rho);
                A[arcos[i]] = matriz;
                matriz = new double[3];
            }
        }
        public void DepositaFerom(Hormiga hormiga, StreamWriter sw)
        {
            sw.WriteLine("La hormiga {0} recorrió los arcos
siguientes, donde se depositó la cantidad indicada:",
hormiga.Visitados[0]);
```

```
            List<string> arcos = new List<string>();
            string arco;
            for (int i = 0; i < hormiga.Visitados.Count - 1; i++)
            {
                if (A.ContainsKey("(" + hormiga.Visitados[i] + "," +
hormiga.Visitados[i + 1] + ")"))
                {
                    arco = "(" + hormiga.Visitados[i] + "," +
hormiga.Visitados[i + 1] + ")";
                }
                else
                {
                    arco = "(" + hormiga.Visitados[i + 1] + "," +
hormiga.Visitados[i] + ")";
                }
                arcos.Add(arco);
            }
            double[] matriz = new double[3];
            for (int i = 0; i < arcos.Count; i++)
            {
                matriz = (double[])A[arcos[i]];
                matriz[1] += 1 / hormiga.Distancia;
                A[arcos[i]] = matriz;
                sw.WriteLine("Arco: {0}   Depósito de feromona: {1}
Feromona total: {2}", arcos[i], 1 / hormiga.Distancia, matriz[1]); //1
/ matriz[0]
                matriz = new double[3];
            }
        }
    }
    class Program
    {
        public static List<int> RellenaX(int nodos)
        {
            List<int> X = new List<int>();
            for (int i = 0; i < nodos; i++)
            {
                X.Add(i + 1);
            }
            return X;
        }
        public static void ObtenerParámetros(ref int n_h,ref int
n_i,ref double tau,ref double alfa,ref double beta,ref double rho,ref
int nodos,ref Gráfica G)
        {
            StreamReader sr = new StreamReader("C:\\Parametros.txt");
            string[] separados;
            string linea = sr.ReadLine();
            linea = sr.ReadLine();
            linea = sr.ReadLine();
            separados = linea.Split('=');
            n_h = int.Parse(separados[1]);
            linea = sr.ReadLine();
            separados = linea.Split('=');
            n_i = int.Parse(separados[1]);
            linea = sr.ReadLine();
            separados = linea.Split('=');
            tau = double.Parse(separados[1]);
```

```
            linea = sr.ReadLine();
            separados = linea.Split('=');
            alfa = double.Parse(separados[1]);
            linea = sr.ReadLine();
            separados = linea.Split('=');
            beta = double.Parse(separados[1]);
            linea = sr.ReadLine();
            separados = linea.Split('=');
            rho = double.Parse(separados[1]);
            linea = sr.ReadLine();
            linea = sr.ReadLine();
            linea = sr.ReadLine();
            separados = linea.Split('=');
            nodos = int.Parse(separados[1]);
            linea = sr.ReadLine();
            linea = sr.ReadLine();
            linea = sr.ReadLine();
            string arco;
            double[] datos = new double[3] { 0, tau, 0 };
            while (linea != null)
            {
                separados = linea.Split('.');
                arco = separados[0];
                datos[0] = double.Parse(separados[1]);
                datos[2] = 1 / datos[0];
                G.A.Add(arco, datos);
                linea = sr.ReadLine();
                datos = new double[3] { 0, tau, 0 };
            }
            sr.Close();
        }
        public static List<int> NumHormigas(int n_h, Gráfica G)
        {
            List<int> nodos_iniciales = new List<int>();
            if(n_h == G.X.Count)
            {
                for(int i=1; i<n_h+1;i++)
                {
                    nodos_iniciales.Add(i);
                }
            }
            else
            {
                Random rn = new Random(DateTime.Now.Millisecond);
                double aleatorio = new double();
                aleatorio = rn.NextDouble();
                for(int i=0; i<n_h;i++)
                {
                    nodos_iniciales.Add((int)Math.Truncate(G.X.Count *
aleatorio + 1));
                    aleatorio = rn.NextDouble();
                }
            }
            return nodos_iniciales;
        }
        static void Main(string[] args)
        {
```

```
            Console.Title = "Sistema de Hormigas para el Agente
Viajero";
            int num_hormigas = new int();
            int num_iteraciones= new int();
            double tau_0 = new double();
            double alpha = new double();
            double beta = new double();
            double rho = new double();
            int nodos = new int();
            StreamWriter sw = new StreamWriter("C:\\Resutados
SH.txt");
            Gráfica G = new Gráfica();
            ObtenerParámetros(ref num_hormigas,ref num_iteraciones,ref
tau_0,ref alpha,ref beta,ref rho,ref nodos,ref G);
            G.X = RellenaX(nodos);
            Hormiga hormiga = new Hormiga();
            Hormiga mejor_solucion = new Hormiga();
            mejor_solucion.Distancia = Double.MaxValue;
            List<Hormiga> recorridos_por_iteración = new
List<Hormiga>();
            Random aleatorio = new Random(DateTime.Now.Millisecond);
            int t = 0;
            List<int> nodos_iniciales = new List<int>();
            nodos_iniciales = NumHormigas(num_hormigas,G);
            while (t < num_iteraciones)
            {
                sw.WriteLine("");
                sw.WriteLine("Iteración {0}", t + 1);
                sw.WriteLine("");
                foreach(int j in nodos_iniciales)
                {
                    hormiga.nodo_actual = j;
                    hormiga.Visitados.Add(hormiga.nodo_actual);
                    sw.WriteLine("La hormiga inicia en el nodo {0}",
j);
                    while (hormiga.Visitados.Count < G.X.Count)
                    {
                        hormiga.Candidatos =
hormiga.ObtenerCandidatos(G.A, G.X, sw);
                        G.X = RellenaX(nodos);
                        Hashtable probas =
hormiga.ObtenerProbabilidades(G.A, alpha, beta, sw);
                        hormiga.nodo_actual =
hormiga.NodoSiguiente(probas, sw, aleatorio);
                        hormiga.Visitados.Add(hormiga.nodo_actual);
                    }
                    hormiga.Visitados.Add(j);
                    sw.WriteLine("Finalmente regresamos al nodo {0}",
j);
                    sw.WriteLine("");
                    hormiga.Distancia = hormiga.ObtenerDistancia(G);
                    hormiga.SolucionObtenida(G, sw);
                    sw.WriteLine("");
                    sw.WriteLine("");
                    recorridos_por_iteración.Add(hormiga);
                    if(mejor_solucion.Distancia > hormiga.Distancia)
                    {
                        mejor_solucion = hormiga;
```

```
                    }
                    hormiga = new Hormiga();
                }
                sw.WriteLine("");
                sw.WriteLine("Evaporamos el rastro de manera global
usando rho = {0}", rho);
                sw.WriteLine("");
                G.EvaporaRastro(rho);
                sw.WriteLine("");
                sw.WriteLine("El deposito de feromonas fue el
siguiente:");
                sw.WriteLine("");
                for (int i = 0; i < recorridos_por_iteración.Count;
i++)
                {
                    G.DepositaFerom(recorridos_por_iteración[i], sw);
                }
                recorridos_por_iteración = new List<Hormiga>();
                t += 1;
            }
            sw.WriteLine();
            sw.WriteLine("Mejor solución");
            Console.WriteLine("Mejor solución");
            sw.Write("Nodos: ");
            Console.Write("Nodos: ");
            foreach(int nodo in mejor_solucion.Visitados)
            {
                sw.Write(nodo);
                Console.Write(nodo);
            }
            sw.WriteLine();
            sw.WriteLine("Con una distancia de: {0}",
mejor_solucion.Distancia);
            Console.WriteLine();
            Console.WriteLine("Con una distancia de: {0}",
mejor_solucion.Distancia);
            sw.Close();
            Console.ReadKey();
        }
    }
}
```

Código para resolver un problema de Clique de Cardinalidad Máxima usando Sistema de Hormigas

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IO;

namespace SH_para_el_Clique_Maximo
{
    public class Hormiga
    {
        public int nodo_actual = new int();
        public List<int> clique = new List<int>();
        public List<int> candidatos = new List<int>();
        public void ObtenerCandidatos(int nodo, Hashtable X,
List<string> A)
        {
            foreach(DictionaryEntry n in X)
            {
                if(A.Contains("("+ n.Key +","+ nodo +")") ||
A.Contains("("+nodo+","+n.Key+")"))
                {
                    candidatos.Add((int)n.Key);
                }
            }
        }
        public void ObtenerCandidatos(int nodo, List<int> Cand,
List<string>A)
        {
            List<int> removidos = new List<int>();
            Cand.Remove(nodo);

            for (int i = 0; i < Cand.Count;i++ )
            {

            }
                for (int n = 0; n < Cand.Count; n++)
                {
                    if (A.Contains("(" + Cand[n] + "," + nodo + ")")
|| A.Contains("(" + nodo + "," + Cand[n] + ")"))
                    {

                    }
                    else
                    {
                        removidos.Add(Cand[n]);
                    }
                }
                for (int i = 0; i < removidos.Count; i++)
                {
                    Cand.Remove(removidos[i]);

                }
        }
```

```
    }
    public class Grafica
    {
        public Hashtable X = new Hashtable();
        public List<string> A = new List<string>();
    }
    class Program
    {
        static void ObtenerParámetros(ref int n_h, ref int n_i, ref
double tau, ref double alfa, ref double beta, ref double rho, ref int
nodos, ref Grafica G)
        {
            StreamReader sr = new StreamReader("C:\\Parametros
Clique.txt");
            string[] separados;
            string linea = sr.ReadLine();
            linea = sr.ReadLine();
            linea = sr.ReadLine();
            separados = linea.Split('=');
            n_h = int.Parse(separados[1]);
            linea = sr.ReadLine();
            separados = linea.Split('=');
            n_i = int.Parse(separados[1]);
            linea = sr.ReadLine();
            separados = linea.Split('=');
            tau = double.Parse(separados[1]);
            linea = sr.ReadLine();
            separados = linea.Split('=');
            alfa = double.Parse(separados[1]);
            linea = sr.ReadLine();
            separados = linea.Split('=');
            beta = double.Parse(separados[1]);
            linea = sr.ReadLine();
            separados = linea.Split('=');
            rho = double.Parse(separados[1]);
            linea = sr.ReadLine();
            linea = sr.ReadLine();
            linea = sr.ReadLine();
            separados = linea.Split('=');
            nodos = int.Parse(separados[1]);
            linea = sr.ReadLine();
            linea = sr.ReadLine();
            linea = sr.ReadLine();
            while (linea != null)
            {
                G.A.Add(linea);
                linea = sr.ReadLine();
            }
            for (int i = 1; i < nodos+1; i++ )
            {
                G.X.Add(i,tau);
            }
                sr.Close();
        }
        static List<int> NumHormigas(int n_h, Grafica G)
        {
            List<int> nodos_iniciales = new List<int>();
            if (n_h == G.X.Count)
```

```
            {
                for (int i = 1; i < n_h + 1; i++)
                {
                    nodos_iniciales.Add(i);
                }
            }
            else
            {
                Random rn = new Random(DateTime.Now.Millisecond);
                double aleatorio = new double();
                aleatorio = rn.NextDouble();
                for (int i = 0; i < n_h; i++)
                {
                    nodos_iniciales.Add((int)Math.Truncate(G.X.Count *
aleatorio + 1));
                    aleatorio = rn.NextDouble();
                }
            }
            return nodos_iniciales;
        }
        static Hashtable ObtenerGrado(List<int> candidatos, Grafica G)
        {
            Hashtable tabla = new Hashtable();
            int contador = 0;
            for (int c = 0; c < candidatos.Count; c++)
            {
                foreach (int n in candidatos)
                {
                    if (G.A.Contains("(" + n + "," + candidatos[c] +
")") || G.A.Contains("(" + candidatos[c] + "," + n + ")"))
                    {
                        contador += 1;
                    }
                }
                tabla.Add(candidatos[c], contador);
                contador = 0;
            }
            return tabla;
        }
        static void CandidatosGradoMayor(ref Hashtable grado)
        {
            Hashtable grado_mayor = new Hashtable();
            bool vacio = false;
            for(int i=grado.Count;i>-1;i--)
            {
                foreach(DictionaryEntry nb in grado)
                {
                    if((int)nb.Value == i)
                    {
                        grado_mayor.Add(nb.Key,nb.Value);
                        vacio = true;
                    }
                }
                if (vacio == true)
                {
                    i = -1;
                }
            }
```

```
            grado = grado_mayor;
        }
        static int CalcularProbabilidades(Grafica G, Hashtable grado,
double alpha, double beta, StreamWriter sw, Random rn)
        {
            Hashtable numeradores = new Hashtable();
            Hashtable probabilidades = new Hashtable();
            double suma = 0;
            foreach(DictionaryEntry nb in G.X)
            {
                if(grado.ContainsKey(nb.Key))
                {
                    numeradores.Add(nb.Key,
Math.Pow((double)nb.Value,alpha));
                    suma += Math.Pow((double)nb.Value, alpha);
                }
            }
            foreach(DictionaryEntry nb in numeradores)
            {
                probabilidades.Add(nb.Key,
(double)nb.Value*Math.Pow(suma,-1));
            }
            foreach(DictionaryEntry nb in probabilidades)
            {
                sw.WriteLine("Nodo {0} Probabilidad de elección {1}",
nb.Key, nb.Value);
            }
            int nodo = new int();
            double aleatorio = rn.NextDouble();
            sw.WriteLine("Número aleatorio {0}", aleatorio);
            double acumulado = 0;
            for (int j = 1; j < G.X.Count + 1; j++)
            {
                if(probabilidades.Contains(j))
                {
                    if (acumulado < aleatorio && aleatorio <=
acumulado + (double)probabilidades[j])
                    {
                        nodo = (int)j;
                    }
                    acumulado += (double)probabilidades[j];
                }
            }
        return nodo;
        }
        static void EvaporaFeromonas(ref Grafica G, double rho)
        {
            for(int i=0; i<G.X.Count;i++)
            {
                G.X[i+1] = (double)G.X[i+1] * (1-rho);
            }
        }
        static void DepositaFermonoas(ref Grafica G, List<Hormiga>
Lista)
        {
            double feromona = new double();
            for (int i = 0; i < Lista.Count; i++)
            {
```

```
                feromona = Math.Pow((G.X.Count + 1 -
Lista[i].clique.Count), -1);
                for (int j = 0; j < Lista[i].clique.Count; j++)
                {
                    G.X[Lista[i].clique[j]] =
(double)G.X[Lista[i].clique[j]] * (1 + feromona);
                }
            }
        }
        static void Main(string[] args)
        {
            Console.Title = "Sistema de Hormigas para el Clique de
Cardinalidad Máxima";
            StreamWriter sw = new StreamWriter("C:\\Resultados SH
Clique de Cardinalidad Máxima.txt");
            int num_hormigas = new int();
            int num_iteraciones = new int();
            double tau_0 = new double();
            double alpha = new double();
            double beta = new double();
            double rho = new double();
            int num_nodos = new int();
            Grafica G = new Grafica();
            ObtenerParámetros(ref num_hormigas,ref num_iteraciones,ref
tau_0,ref alpha,ref beta,ref rho,ref num_nodos,ref G);
            List<int> distr_hormigas = new List<int>();
            Hormiga hormiga = new Hormiga();
            List<Hormiga> Lista_de_hormigas = new List<Hormiga>();
            int t = 0;
            Hormiga mejor_solución = new Hormiga();
            Hashtable grado = new Hashtable();
            Random rn = new Random(DateTime.Now.Millisecond);
            while(t<num_iteraciones)
            {
                distr_hormigas = NumHormigas(num_hormigas, G);
                sw.WriteLine("      Iteración {0}", t+1);
                sw.WriteLine();
                sw.WriteLine("Las hormigas iniciarán en los siguientes
nodos");
                foreach(int nodo in distr_hormigas)
                {
                    sw.Write(nodo +"    ");
                }
                sw.WriteLine();
                sw.WriteLine();
                for (int i = 0; i < distr_hormigas.Count; i++)
                {
                    hormiga.clique.Add(distr_hormigas[i]);
                    hormiga.ObtenerCandidatos(distr_hormigas[i], G.X,
G.A);
                    sw.WriteLine("Hormiga {0} Inicia en el nodo
{1}",i+1, distr_hormigas[i]);
                    sw.WriteLine("Que es adyacente a los nodos:");
                    foreach(int n in hormiga.candidatos)
                    {
                        sw.Write(n+"    ");
                    }
                    sw.WriteLine();
```

```
                        while (hormiga.candidatos.Count > 0)
                        {
                            grado = ObtenerGrado(hormiga.candidatos,
G);
                            foreach (DictionaryEntry nb in grado)
                            {
                                sw.WriteLine("El nodo {0} tiene grado
{1} con respecto a los candidatos", nb.Key, nb.Value);
                            }
                            CandidatosGradoMayor(ref grado);
                            sw.WriteLine("Los de mayor grado son:");
                            foreach (DictionaryEntry nb in grado)
                            {
                                sw.WriteLine("El nodo {0} tiene grado
{1} con respecto a los candidatos", nb.Key, nb.Value);
                            }
                            hormiga.nodo_actual =
CalcularProbabilidades(G, grado, alpha, beta, sw, rn);
                            sw.WriteLine();
                            sw.WriteLine("De donde elegimos al nodo
{0} para anexarlo al clique", hormiga.nodo_actual);
                            sw.WriteLine();
                            hormiga.clique.Add(hormiga.nodo_actual);

hormiga.ObtenerCandidatos(hormiga.nodo_actual, hormiga.candidatos,
G.A);
                            sw.WriteLine("Quedando disponibles los
nodos:");
                            foreach (DictionaryEntry nb in grado)
                            {
                                sw.WriteLine("{0} con grado {1} con
respecto a los candidatos restantes", nb.Key, nb.Value);
                            }
                        }
                    sw.WriteLine();
                    sw.WriteLine("La solución construida es el
clique:");
                    for (int p = 0; p < hormiga.clique.Count;p++ )
                    {
                        sw.Write(hormiga.clique[p]+"    ");
                    }
                    sw.WriteLine();
                    if(mejor_solución.clique.Count <
hormiga.clique.Count)
                    {
                        mejor_solución = hormiga;
                    }
                    Lista_de_hormigas.Add(hormiga);
                    hormiga = new Hormiga();
                    grado = new Hashtable();
                }
                EvaporaFeromonas(ref G, rho);
                DepositaFermonoas(ref G, Lista_de_hormigas);
                sw.WriteLine("Actualizamos y evaporamos las feromonas
quedando el siguiente resultado:");
                foreach(DictionaryEntry nodo in G.X)
                {
```

```
                    sw.WriteLine("Nodo {0} Rastro{1}", nodo.Key,
nodo.Value);
                }
                Lista_de_hormigas = new List<Hormiga>();
                t += 1;
            }
            sw.WriteLine();
            sw.WriteLine();
            Console.WriteLine("El clique encontrado tiene cardinalidad
{0} y es el siguiente:", mejor_solución.clique.Count);
            sw.WriteLine("El clique encontrado tiene cardinalidad {0}
y es el siguiente:", mejor_solución.clique.Count);
            foreach(int n in mejor_solución.clique)
            {
                Console.Write(n);
                sw.Write(n);
            }
            sw.Close();
            Console.ReadKey();
        }
    }
}
```

Código para resolver un problema de Número Cromático usando Recocido Simulado

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Collections;
using System.IO;

namespace RS_para_Num_Cromatico
{
    class Solucion
    {
        public List<Nodo> solucion = new List<Nodo>();
        public int total_colores = new int();
        public bool factible = true;

        public void SolInicial(Grafica G)
        {
            Nodo n = new Nodo();
            for (int i = 0; i < G.X.Count; i++)
            {
                n.nodo = G.X[i];
                n.color = i + 1;
                solucion.Add(n);
                n = new Nodo();
            }
            factible = true;
            TotalDeColores();

        }

        public void TotalDeColores()
        {
            List<int> colores = new List<int>();
            foreach(Nodo n in solucion)
           {
                if(colores.Contains(n.color))
                {

                }
                else
                {
                    colores.Add(n.color);
                }
           }
            total_colores = colores.Count();

        }

    }
    class Nodo
    {
        public int nodo = new int();
        public int color = new int();
    }
```

```
    class Grafica
    {
        public List<int> X = new List<int>();
        public List<string> A = new List<string>();

        public void ObtenerParámetros(StreamReader sr, string linea)
        {
            string[] separados;
            linea = sr.ReadLine();
            separados = linea.Split('=');
            int num_nodos = int.Parse(separados[1]);
            for(int i =1; i<num_nodos+1; i++)
            {
                X.Add(i);
            }
            linea = sr.ReadLine();
            separados = linea.Split(':');
            for(int i =0; i<separados.Length; i++)
            {
                A.Add(separados[i]);
            }

        }
    }
    class Program
    {
        static void ObtenerParametros(ref double T_0,ref double
alfa,ref int N,ref int T_max, Grafica G, StreamReader sr)
        {
            string[] sep;
            string linea = sr.ReadLine();
            G.ObtenerParámetros(sr, linea);
            linea = sr.ReadLine();
            linea = sr.ReadLine();
            linea = sr.ReadLine();
            sep = linea.Split('=');
            T_0 = double.Parse(sep[1]);
            linea = sr.ReadLine();
            sep = linea.Split('=');
            alfa = double.Parse(sep[1]);
            linea = sr.ReadLine();
            sep = linea.Split('=');
            N = int.Parse(sep[1]);
            linea = sr.ReadLine();
            sep = linea.Split('=');
            T_max = int.Parse(sep[1]);
        }

        static void Vecino(Grafica G, ref Solucion s_vecino,Solucion
s_actual, Random rn)
        {
            s_vecino = new Solucion();
            int nodo = new int();
            int color = new int();
            double intervalo = 0;
            double aleatorio = rn.NextDouble();
            for(int i =0; i<G.X.Count;i++)
            {
```

```
                if (aleatorio > intervalo && aleatorio <= intervalo +
Math.Pow(G.X.Count, -1))
                {
                    nodo = G.X[i];
                    i = G.X.Count;
                }
                else
                {
                    intervalo += Math.Pow(G.X.Count, -1);
                }
            }
            aleatorio = rn.NextDouble();
            intervalo = 0;
            for(int i =0; i<s_actual.solucion.Count; i++)
            {
                if (aleatorio > intervalo && aleatorio <= intervalo +
Math.Pow(s_actual.solucion.Count, -1))
                {
                    color = s_actual.solucion[i].color;
                    i = s_actual.solucion.Count;
                }
                else
                {
                    intervalo += Math.Pow(s_actual.solucion.Count, -1);
                }
            }
            Nodo p = new Nodo();
            for (int i = 0; i < s_actual.solucion.Count; i++)
            {
                p.nodo = s_actual.solucion[i].nodo;
                if (s_actual.solucion[i].nodo == nodo)
                {
                    p.color = color;
                }
                else
                {
                    p.color = s_actual.solucion[i].color;
                }
                s_vecino.solucion.Add(p);
                p = new Nodo();
            }
            s_vecino.TotalDeColores();
            nodo = new int();
            color = new int();
            aleatorio = new double();
        }

        static double FuncionObjetivo(Solucion s, Grafica G)
        {
            List<int> matriz = new List<int>();
            double z = new double();
            for (int i = 0; i < s.solucion.Count; i++ )
            {
                if (matriz.Contains(s.solucion[i].color))
                {
                }
                else
                {
```

```
                    matriz.Add(s.solucion[i].color);
                }
            }
            int contador = 0;
            int arcos_malos = 0;
            List<int> lista = new List<int>();
            for (int i = 0; i < matriz.Count; i++)
            {
                for(int j=0; j<s.solucion.Count; j++)
                {
                    if(s.solucion[j].color == matriz[i])
                    {
                        contador += 1;
                        lista.Add(s.solucion[j].nodo);
                    }
                }
                for (int j = 0; j < lista.Count;j++ )
                {
                    for(int e = j; e<lista.Count; e++)
                    {
                        if (G.A.Contains("(" + lista[j] + "," +
lista[e] + ")") || G.A.Contains("(" + lista[e] + "," + lista[j] + ")"))
                        {
                            arcos_malos += 1;
                            s.factible = false;
                        }
                    }
                }
                z += Math.Pow(contador, 2);
                z -= 2 * contador * arcos_malos;
                contador = 0;
                arcos_malos = 0;
                lista = new List<int>();
            }
            return z;
        }

        static bool SolucionesIguales(Solucion s1, Solucion s2)
        {
            bool iguales = true;
            for (int i = 0; i < s1.solucion.Count;i++ )
            {
                if (s1.solucion[i].color == s2.solucion[i].color)
                {

                }
                else
                {
                    iguales = false;
                }
            }
                return iguales;
        }

        static void Main(string[] args)
        {
            StreamWriter sw = new StreamWriter("D:\\Resutados RS
NC.txt");
```

```
            StreamReader sr = new StreamReader("D:\\Parámetros
RS.txt");
            Grafica G = new Grafica();
            double T_0 = new double();
            double alfa = new double();
            int N = new int();
            int T_max = new int();

            ObtenerParametros(ref T_0,ref alfa, ref N, ref T_max, G,
sr);
            sr.Close();
            Solucion s_actual = new Solucion();
            Solucion s_mejor = new Solucion();
            Solucion s_vecino = new Solucion();
            Random rn = new Random(DateTime.Now.Millisecond);
            int n = 0;
            int t = 0;
            Solucion s_0 = new Solucion();
            s_0.SolInicial(G);
            sw.WriteLine("Solución Inicial");
            foreach(Nodo s in s_0.solucion)
            {
                sw.Write("({0},{1})",s.nodo,s.color);
            }
            sw.WriteLine();
            s_mejor = s_0;
            s_actual = s_0;
            double temp = T_0;
            sw.WriteLine(@"Temperatura: {0}
",temp);

            while (t < T_max)
            {
             sw.WriteLine(@"    Iteración: {0}
", t+1);
             sw.WriteLine("Solución actual: ");
             foreach (Nodo s in s_actual.solucion)
             {
                 sw.Write("({0},{1})", s.nodo, s.color);
             }
             sw.WriteLine();
            Vecino(G, ref s_vecino, s_actual, rn);
            while (SolucionesIguales(s_vecino, s_actual))
            {
                Vecino(G, ref s_vecino, s_actual, rn);
            }
            sw.WriteLine("Solución Vecina Construida:");
            foreach (Nodo s in s_vecino.solucion)
            {
                sw.Write("({0},{1})", s.nodo, s.color);
            }
            sw.WriteLine();
            sw.WriteLine("Calculamos el valor de la Función
Objetivo:");
            sw.WriteLine("z(s_act) = {0}", FuncionObjetivo(s_actual,
G));
            sw.WriteLine("z(s_vec) = {0}", FuncionObjetivo(s_vecino,
G));
```

```
            if(FuncionObjetivo(s_actual,G) <
FuncionObjetivo(s_vecino,G))
            {
                sw.WriteLine("Como el vecino es mejor lo seleccionamos
automáticamente y lo hacemos s_actual");
                s_actual = s_vecino;
            }
            else
            {
                sw.WriteLine("Como este vecino no mejora la función
objetivo generamos un número aleatorio R y la probabilidad de
elección");
                double aleatorio = rn.NextDouble();
                sw.WriteLine("R = {0}",aleatorio);
                sw.WriteLine("Probabilidad: {0}", Math.Pow(Math.E, -
Math.Abs(FuncionObjetivo(s_actual, G) - FuncionObjetivo(s_vecino,
G))*Math.Pow(temp,-1)));
                if (aleatorio < Math.Pow(Math.E, -
Math.Abs(FuncionObjetivo(s_actual, G) - FuncionObjetivo(s_vecino, G)) *
Math.Pow(temp, -1)))
                {
                    s_actual = s_vecino;
                    sw.WriteLine("Como R es menor, elegimos a s_vecino
aunque sea peor");
                }
            }
            n += 1;
            sw.WriteLine("n = {0}", n);
            if(n== N)
            {
                temp *= alfa;
                sw.WriteLine("Actualizamos la temperatura a: {0}",
temp);
                n = 0;
            }
            if (s_actual.factible == true && FuncionObjetivo(s_mejor,
G) < FuncionObjetivo(s_actual, G))
            {
                s_mejor = s_actual;
                sw.WriteLine("*** Al ser s_actual factible y mejor la
reemplazamos ***");
            }
            t += 1;
            if(t == T_max)
            {
                Console.WriteLine("Hemos terminado");
                Console.WriteLine("Número mínimo para colorear la
gráfica {0}",s_mejor.total_colores);
                Console.WriteLine(s_mejor.factible);
                Console.WriteLine("Total de Iteraciones: {0}", t);
                sw.WriteLine("Mejor Solución:");
                foreach (Nodo s in s_mejor.solucion)
                {
                    sw.Write("({0},{1})", s.nodo, s.color);
                }
                sw.WriteLine();
                sw.WriteLine("Número mínimo de colores:
{0}",s_mejor.total_colores);
```

```
                }
                }
                sw.Close();
                Console.ReadKey();
            }
        }
    }
```