BENEMÉRITA

UNIVERSIDAD AUTÓNOMA DE PUEBLA

FACULTAD DE CIENCIAS FÍSICO MATEMÁTICAS

“APLICACIÓN DE ALGUNAS HEURÍSTICAS AL PROBLEMA DE LA MOCHILA”

TESIS

PARA OBTENER EL TÍTULO DE:

LICENCIADO EN MATEMÁTICAS APLICADAS

PRESENTA:

GERMÁN ANTONIO VÁZQUEZ ROMERO

ASESOR:

DRA. LIDIA AURORA HERNÁNDEZ REBOLLAR

CO-ASESOR:

DRA.MARÍA DE LOURDES SANDOVAL SOLÍS

18 DE DICIEMBRE DE 2013

DEDICATORIA

Le agradezco al alfa y la omega , es decir a dios que es el principio y fin de todas las cosas, por haberme acompañado y guiado a lo largo de mi carrera, por ser mi fortaleza en momentos de debilidad y por brindarme una vida llena de aprendizajes, experiencias y sobre todo de felicidad.

Le doy gracias a mis padres Jesús Antonio y Cruz por apoyarme en todo momento, por los valores que me han inculcado, y por haberme dado la oportunidad de tener una excelente educación en el transcurso de mi vida. Sobre todo por ser un excelente ejemplo de vida a seguir.

A mis hermanos Ricardo y José de Jesús por ser parte importante de mi vida y representar la unidad familiar, por ser un ejemplo de trabajo y desarrollo profesional a seguir.

A Ana Aleyda por ser una parte importante de mi vida, por haberme apoyado en las buenas y en las malas, sobre todo por su paciencia y amor incondicional.

Les agradezco la confianza, apoyo y dedicación de tiempo a todos mis profesores. Por haber compartido conmigo sus conocimientos y sobre todo su amistad a lo largo de mi carrera.

Gracias Dra. Lidia Aurora Hernández Rebollar y Dra. María de Lourdes Sandoval Solís por creer en mí, y haberme brindado la oportunidad de desarrollar mi tesis profesional con usted(es) y por todo el apoyo y facilidades que me fueron otorgadas. Por darme la oportunidad de crecer profesionalmente y aprender nuevas cosas.

También quiero agradecer a mis revisores de tesis y sinodales que, sin su apoyo, tiempo y dedicación no me hubiera sido posible concluir mi tesis.

A mis amigos por confiar y creer en mí y haber hecho de mi etapa universitaria un trayecto de vivencias que nunca olvidare, pero sobre todo por llenar mi vida de alegrías cuando más lo he necesitado.

AGRADECIMIENTOS

Le agradezco al alfa y la omega , es decir a dios que es el principio y fin de todas las cosas, por haberme acompañado y guiado a lo largo de mi carrera, por ser mi fortaleza en momentos de debilidad y por brindarme una vida llena de aprendizajes, experiencias y sobre todo de felicidad.

Le doy gracias a mis padres Jesús Antonio y Cruz por apoyarme en todo momento, por los valores que me han inculcado, y por haberme dado la oportunidad de tener una excelente educación en el transcurso de mi vida. Sobre todo por ser un excelente ejemplo de vida a seguir.

A mis hermanos Ricardo y José de Jesús por ser parte importante de mi vida y representar la unidad familiar, por ser un ejemplo de trabajo y desarrollo profesional a seguir.

A Ana Aleyda por ser una parte importante de mi vida, por haberme apoyado en las buenas y en las malas, sobre todo por su paciencia y amor incondicional.

Les agradezco la confianza, apoyo y dedicación de tiempo a todos mis profesores. Por haber compartido conmigo sus conocimientos y sobre todo su amistad a lo largo de mi carrera.

Gracias Dra. Lidia Aurora Hernández Rebollar y Dra. María de Lourdes Sandoval Solís por creer en mí, y haberme brindado la oportunidad de desarrollar mi tesis profesional con usted(es) y por todo el apoyo y facilidades que me fueron otorgadas. Por darme la oportunidad de crecer profesionalmente y aprender nuevas cosas.

También quiero agradecer a mis revisores de tesis y sinodales que, sin su apoyo, tiempo y dedicación no me hubiera sido posible concluir mi tesis.

A mis amigos por confiar y creer en mí y haber hecho de mi etapa universitaria un trayecto de vivencias que nunca olvidare, pero sobre todo por llenar mi vida de alegrías cuando más lo he necesitado.

INTRODUCCIÓN

Dentro del área de la algoritmia y programación, *el problema de la mochila,* comúnmente abreviado por KP (del inglés *Knapsack problem*) es un problema de optimización entera. Modela una situación análoga al llenar una mochila, capaz de soportar un peso máximo, con todo o parte de un conjunto de objetos, cada uno con un peso y valor específicos. Los objetos colocados en la mochila deben maximizar el valor total sin exceder el peso máximo.

Si bien la formulación del problema es sencilla, su solución es compleja. La estructura única del problema lo convierten en un problema frecuente en la investigación.

Para este problema, consideremos objetos diferentes, cada uno de ellos tiene un costo y un peso asignados al i-ésimo objeto, el peso máximo o capacidad soportada por la mochila es Se asume que los valores y pesos no son negativos. Para simplificar la representación, se suele asumir que los objetos están listados según su peso en orden creciente.

La formulación más común del problema, es la del llamado *problema de la mochila 0-1,* que restringe el número de copias de cada tipo de objetos a cero o uno. Matemáticamente, el problema queda formulado de la siguiente manera (como un problema de *programación lineal):*

Este problema se ha resuelto tradicionalmente mediante los métodos de *programación lineal entera,* descritos más adelante. El hecho de que se trate de un problema de programación lineal hace referencia a que la función a optimizar y las inecuaciones que constituyen las restricciones son lineales.

El problema de la mochila, puede resolverse a través de los denominados *algoritmos heurísticos.* En programación, es fundamental encontrar algoritmos con buenos tiempos de ejecución y buenas soluciones. Una heurística normalmente encuentra buenas soluciones o se ejecuta razonablemente rápido, aunque no existe tampoco prueba de que siempre será así. Las heurísticas generalmente son usadas cuando existe una solución óptima bajo las restricciones dadas tiempo, espacio, etc [12].

A menudo, pueden encontrarse instancias concretas del problema donde la heurística producirá resultados muy malos o se ejecutará muy lentamente. Aun así, estas instancias concretas pueden ser ignoradas porque no deberían ocurrir nunca en la práctica por ser de origen teórico. Por lo tanto, el uso de heurísticas es muy común en el mundo real [12]. Es por esto que nos planteamos el siguiente objetivo:

Comparar algunos algoritmos heurísticos programados en Matlab, tanto en tiempo de ejecución, número de iteraciones y la mejor solución para el problema de la mochila.

El contenido de esta tesis consta de cuatro capítulos que abarcan una breve panorámica de la programación lineal, programación entera, algunos algoritmos heurísticos y la implementación computacional.

En el primer capítulo se han incluido algunos problemas clásicos de la programación lineal como punto de partida para su estudio, que corresponden en gran medida a los problemas que históricamente dieron origen a esta disciplina. En este capítulo también se desarrolla el estudio de las distintas formas equivalentes en que usualmente se presentan los problemas de programación lineal, también se incluye la presentación un tanto informal del método simplex.

En el segundo capítulo se presentan algunos problemas clásicos de la programación entera. Además, se presenta un estudio introductorio de los

conocidos métodos planos de corte, ramificación y acotación, etc., para la solución de problemas enteros de la programación lineal.

El tercer capítulo incluye un estudio de los principales conceptos sobre las técnicas heurísticas en las que está basada la solución al problema de la mochila, fundamentalmente una panorámica de los algoritmos hill climbing (colina inclinada), backtracking (búsqueda hacia atrás), simulated annealing (recocido simulado) y tabu search (búsqueda tabu). Su estudio lo he considerado fundamental puesto que de estos algoritmos depende el que se adquiera la intuición necesaria para abordar la solución de problemas tipo mochila.

En el cuarto capítulo aparece la implementación de los algoritmos heurísticos, los cuales están programados en MATLAB y cuyo objetivo es comparar cuál de los algoritmos heurísticos es el mejor, tanto en tiempo de ejecución, como el número de corridas así como el algoritmo que da la mejor solución óptima para el problema de la mochila.

Índice general

“APLICACIÓN DE ALGUNAS HEURÍSTICAS

AL PROBLEMA DE LA MOCHILA”

GERMÁN ANTONIO VÁZQUEZ ROMERO

DICIEMBRE DE 2013.

CAPÍTULO 1 BREVE HISTORIA DE LA PROGRAMACIÓN LINEAL Y SUS APLICACIONES

La programación lineal estudia el problema de minimizar o maximizar una función lineal en la presencia de desigualdades lineales. Desde que George B. Dantzig desarrolló el método simplex en 1947, la programación lineal se ha utilizado extensamente en el área militar, industrial, gubernamental y de planificación urbana, entre otras [7]. La popularidad de la programación lineal se puede atribuir a muchos factores, incluyendo su habilidad para modelar problemas grandes y complejos, y la habilidad de los usuarios para resolver problemas a gran escala en un intervalo de tiempo razonable mediante el uso del método simplex y de computadoras.

A partir de la Segunda Guerra Mundial se hizo evidente que era esencial la planificación y coordinación entre varios proyectos, así como el uso eficaz de los recursos disponibles. En junio de 1947 se inició un trabajo intensivo del equipo de la Fuerza Aérea de los EE.UU. conocido como SCOOP (Scientific Computation of Optimum Programs). Como resultado, George B. Dantzig desarrolló el método simplex para el final del verano de 1947. El interés de la programación lineal se difundió rápidamente entre economistas, matemáticos, estadísticos e instituciones gubernamentales.

Desde la creación del método simplex mucha gente ha contribuido al crecimiento de la programación lineal, ya sea desarrollando su teoría matemática, diseñando códigos y métodos computacionales eficientes, experimentando nuevas aplicaciones, y también utilizando la programación lineal como una herramienta auxiliar para resolver problemas más complejos como son programas enteros, programas discretos, programas no lineales, problemas combinatorios, problemas de programación estocástica y problemas de control óptimo.

La habilidad para establecer objetivos generales y después encontrar políticas de soluciones óptimas a problemas prácticos de decisión de gran complejidad es un desarrollo revolucionario. En ciertos campos tales como la planeación en las industrias del petróleo y química, la programación lineal ha encontrado un amplio uso para obtener la minimización de costos [7].

En otros campos, como es modelando la dinámica del crecimiento poblacional del mundo contra una base de recursos decrecientes, se ha apreciado escasamente su potencia para elevar los estándares de vida [19].

1.1 El problema de la programación lineal

Un problema de programación lineal es un problema de minimizar o maximizar una función lineal en la presencia de restricciones lineales del tipo desigualdad, igualdad o ambas. En esta sección se formula el problema de programación lineal.

Definiciones básicas

En general, un problema de programación lineal puede formularse como sigue.

. . . .

Aquí es la *función objetivo* que debe minimizarse y se denotará por *z*. Los coeficientes son los coeficientes de costo (conocidos), y son las variables de decisión que deben determinarse. La desigualdad denota la *i*-ésima restricción. Los coeficientes

para *i=1,2,..., m, j=1,2,…, n* se llaman *coeficientes tecnológicos.* Estos coeficientes tecnológicos forman la *matriz de restricciones* siguiente:

El vector columna cuya *i-ésima componente es* , el cual se denomina *vector del lado derecho,* representa los requerimientos mínimos que deben satisfacerse. Las restricciones son las *restricciones de no negatividad.* Un conjunto de variables que satisfacen todas las restricciones se denomina *punto factible o vector factible*. El conjunto de todos estos puntos se llama *región factible o espacio factible.*

Usando la terminología anterior, el problema de programación lineal se puede enunciar como sigue: Entre todos los vectores factibles encuéntrese aquel que minimiza (o maximiza) la función objetivo.

Ejemplo 1.1.1

Considere el siguiente problema lineal.

En este caso se tienen dos variables de decisión y . La función objetivo que debe minimizarse es . En la figura 1.1.1 se ilustran las restricciones y la región factible.

Así pues, el problema de optimización consiste en encontrar un punto en la región factible que da el valor mínimo de la función objetivo.

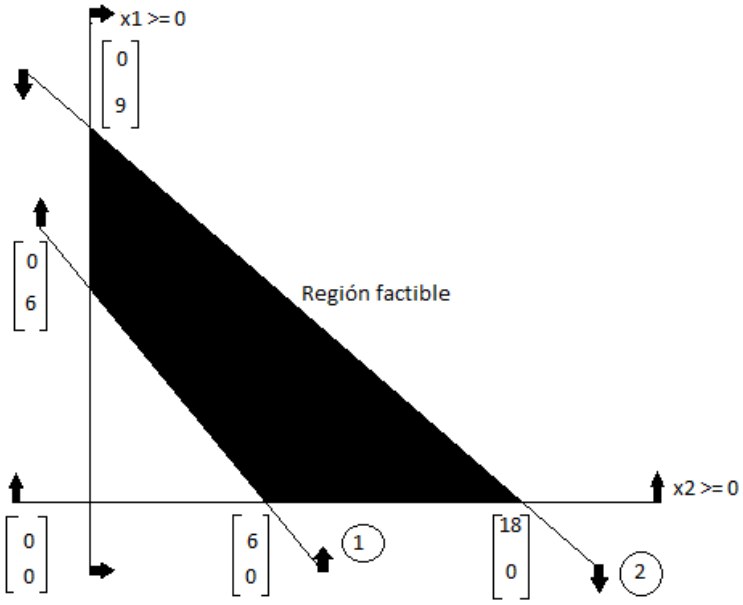

Figura 1.1.1 ilustración de la región factible.

La programación lineal en forma matricial

Usando la notación matricial es posible plantear en forma más conveniente un problema de programación lineal. Por ejemplo, considérese el siguiente problema.

Sea el vector renglón y considérense los vectores columna y , y la matriz de dados por:

Entonces el problema anterior se puede escribir como.

El problema también se puede representar convenientemente usando las columnas de . Escribiendo como en donde es la -ésima columna de , el problema se puede plantear como sigue:

1.2 Ejemplos de problemas lineales

En esta sección se describen varios problemas que se pueden formular como programas lineales. El propósito es mostrar la gran variedad de problemas que se pueden reconocer y expresar como programas lineales en términos matemáticos precisos.

Problema de mezcla alimentaria

Un molino agrícola produce alimento para pollos misma que se hace mezclando varios ingredientes, la mezcla debe hacerse de tal manera que el alimento satisfaga ciertos niveles para diferentes tipos de nutrientes, como son proteínas, calcio, carbohidratos y vitaminas. Para ser más específico, supóngase que se consideran ingredientes y nutrientes . Supóngase, que el costo por unidad del ingrediente es y que la cantidad que se usará del ingrediente es . El costo es, por lo tanto, Si la cantidad requerida del

producto final es , entonces se debe tener que Supóngase, además, que la cantidad del nutriente presente en una unidad del ingrediente es y que los límites aceptables, inferior y superior, del nutriente en una unidad del alimento para pollos son y , respectivamente. Entonces, se debe tener las restricciones para .

Finalmente, supóngase que debido a un periodo de escasez, el molino no puede adquirir más de unidades del ingrediente . El problema de mezclar los ingredientes de tal manera que el costo sea mínimo y las restricciones anteriores se satisfagan, se puede formular como sigue.

. . . .

. . . .

El problema del transporte

Considérense puntos de origen localizados en un mapa, donde el origen tiene una provisión de unidades de un cierto artículo. Además, están localizados puntos de destino, en donde el destino requiere de unidades del producto. Supóngase que Asociado con cada ligadura o arco del origen al destino , se tiene un costo unitario por transporte. El problema consiste en

determinar el patrón de producción-transporte que minimice el costo total de transporte.

Sea el número de unidades transportadas a lo largo del arco del origen al destino . Supóngase, además, que la oferta total es igual a la demanda total, es decir,

Si la oferta total excede a la demanda total, entonces se puede introducir un destino ficticio o artificial con demanda , y para suponiendo que la oferta total es igual a la demanda total, el modelo de programación lineal para el problema del transporte resulta ser el siguiente:

En la figura 1.2.2 se ilustra gráficamente el problema del transporte.

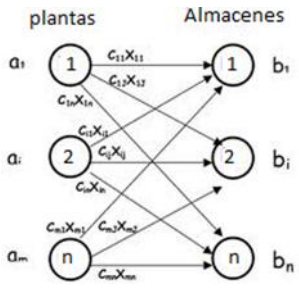

Figura 1.2.2 ilustración gráfica de un problema del transporte

El problema del transporte se puede escribir en forma matricial si se hace

En donde

Y y son vectores unitarios de de con unos en la -ésima y la -ésima posiciones, respectivamente. Obsérvese que es un escalar, es un vector. Con estas definiciones el problema toma la siguiente forma:

La matriz , con dimensiones tiene la siguiente forma especial.

columnas

renglones

En donde es un -vector renglón de componentes iguales a 1, e es una matriz identidad de . La matriz es la que da al problema del transporte su estructura especial.

Ejemplo 1.2.2

Como un ejemplo de un problema del transporte considérese un problema con 2 orígenes, 3 destinos y con los datos siguientes:

		destino			
		1	2	3	
origen	1				30
	2				20
		15	10	25	

La matriz Para este problema del transporte viene dada por:

A continuación se introducen las soluciones básicas factibles, puesto que existe una caracterización algebraica de tales soluciones, será posible ir de una solución básica factible a otra, hasta alcanzar la optimalidad

Definición (soluciones básicas factibles)

Considérese el sistema y en donde es una matriz de y es un vector. Supóngase que rango Después de un posible rearreglo de las columnas de , sea en donde es la matriz invertible de y es una matriz de El punto , en donde:

Se llama *solución básica* del sistema. Si entonces se llama *solución básica factible* del sistema. Aquí, se llama *matriz básica* y se llama la *matriz*

no básica. Las componentes de se llaman *variables básicas*, y las componentes de se llaman *variables no básicas*. Si entonces se llama *solución básica factible no degenerada,* y si al menos una componente de es cero, entonces se llama *solución básica factible degenerada.*

1.3 El método simplex

A continuación se dará un resumen del método simplex para resolver el siguiente problema de programación lineal:

EL algoritmo simplex (problema de minimización)

PASO INICIAL

Selecciónese una solución básica factible inicial donde se llama la matriz básica o simplemente la base.

PASO PRINCIPAL

1. Resuélvase el sistema (con solución única). Sea y
2. Resuélvase el sistema (con solución única). Para todas las variables no básicas calcúlese Sea en donde es el conjunto de índices asociado con las variables no básicas. Si entonces el proceso se detiene con la solución básica factible presente como solución óptima. En caso contrario, se sigue al paso 3.
3. Resuelva el sistema (con solución única). Si , entonces el proceso se detiene con la conclusión de que la solución óptima es no acotada a lo largo del rayo en donde es un

vector de ceros excepto por un 1 en la -ésima posición. Si , se sigue al paso 4.

4. Ahora, entra a la base y la variable de bloqueo sale de la base en donde el índice se determina mediante el siguiente criterio de la razón mínima:

— —

Se actualiza la base en donde reemplaza a , se actualiza el conjunto de índices, donde el nuevo punto es — y se repite el paso1.

CAPÍTULO 2 PROGRAMACIÓN ENTERA

En la programación lineal que se estudia comúnmente hay muchos problemas prácticos, en los cuales las variables de decisión sólo tienen sentido real si su valor es entero, con frecuencia es necesario asignar a las actividades cantidades enteras de personas, máquinas o vehículos. El hecho de exigir valores enteros es la única diferencia que tiene un problema con la formulación de programación lineal, entonces se trata de un problema de programación entera.

En este capítulo se presentan algunos métodos de solución de problemas enteros, es decir, aquellos cuyas variables de decisión no pueden tomar valores fraccionarios.

En casi todos los campos de la ciencia, incluyendo la optimización, hay un cierto abismo entre la teoría y la práctica. En los métodos de programación entera este abismo es bastante profundo. Aunque muchos de los métodos han demostrado teóricamente su convergencia a una solución óptima entera, en la práctica, esta convergencia, puede resultar tan lenta, que para fines prácticos el método resulta inservible, es decir, es muy caro en términos del tiempo requerido de uso de una computadora.

Parte del problema en la programación entera radica en la diferencia esencial que existe entre ésta y la programación lineal. En la programación lineal se maximiza o minimiza una función sobre una región de factibilidad convexa, mientras que en la programación entera se maximiza una función sobre una región de factibilidad que generalmente no es convexa. Por lo tanto, la solución de problemas enteros, es de muchos órdenes de magnitud más complicada que la programación lineal.

Es importante aclarar que los métodos aquí presentados resumen el estado actual de la programación entera, pero que estos métodos distan bastante de ser cien por ciento eficientes en la solución de todos los problemas enteros (que es el caso contrario a otros métodos de optimización que dan la solución óptima para todo problema).

Los métodos de programación entera, *no aseguran* que un problema entero de tamaño regular, pueda ser resuelto en un tiempo razonable en una computadora. Por lo tanto los algoritmos seleccionados en este capítulo obedecen a los siguientes criterios:

a) *históricos,* pues representaron un avance en la teoría al ser descubiertos.
b) *Prácticos,* pues resuelven eficientemente ciertos problemas enteros.
c) *Vanguardia,* pues representan la materia actual de investigación que probablemente abrirá brecha en el futuro.

Se inicia este capítulo ilustrando ejemplos de diferentes problemas de naturaleza entera. A continuación, se exploran algunos métodos basados en *planos de corte,* que no son eficientes para resolver problemas de tamaño modesto, pero históricamente fueron los primeros en abrir la brecha en esta área. A continuación se presentan métodos de *enumeración implícita* que trabajan adecuadamente para problemas del tipo binario (cero-uno) con un máximo de 130 variables binarias y métodos de *ramificación y acotación,* que aparentemente son los que se comportan mejor en la solución de problemas enteros de dimensión intermedia. Posteriormente se presentan una serie de *algoritmos heurísticos* para resolver problemas de carácter combinatorio, los cuales se caracterizan por tener un conjunto de posibles soluciones discretas tales como *el problema de la mochila.*

Se discute en el presente capítulo la información concerniente a la experiencia numérica por computadora, en la solución de estos problemas enteros [22].

2.1 Ejemplos ilustrativos

Un programa entero es sencillamente el modelo de programación lineal con la restricción adicional de que las variables deben tener valores enteros. Los tres problemas de estructura entera son

Problema entero (PE)

Problema entero-mixto (PEM)

Problemas entero-cero-uno (PECU) o problema binario (PB)

Estos tres tipos de problemas requieren de técnicas especiales de solución, ya que los métodos de solución de los programas lineales (*simplex, dual simplex, revisado, etc.),* por lo general, no siempre obtienen la solución óptima.

Ejemplo 2.1.1 dado el siguiente problema entero,

si se resuelve por el *método simplex*, ignorando las restricciones enteras, se obtiene la solución redondeando al entero más cercano

quedaría Sin embargo, esta solución viola las restricciones, porque

El tratar de redondear al entero inmediato menor o mayor crea el problema combinatorio, de que si hay variables de decisión, se deben analizar diferentes posibilidades. Por lo tanto, si hay 100 variables, el número de posibilidades a analizar es del orden de Consecuentemente se requieren técnicas más eficientes que el análisis exhaustivo de todas las alternativas posibles.

Se hace notar que si al resolver un problema entero (PE), por medio de alguna técnica de programación lineal, el resultado óptimo del problema lineal (PL) (generado por el problema entero (PE), al ignorar las restricciones de integralidad de las variables del mismo) es entero, entonces es también una solución óptima del problema entero original (PE).

Se ve a continuación qué variedad de problemas caen dentro de esta familia de problemas enteros.

a) *Todos los problemas de programación lineal, donde las actividades, por su estructura deben ser no-divisibles, son programas enteros.*
   Por ejemplo, problemas de producción de automóviles, prendas de vestir, etc. ¿Qué significado tendría la producción de 577.83 automóviles?
b) *Todos los problemas de asignación, redes de optimización y algunos problemas de transporte,* son problemas enteros. Sin embargo, dada la estructura tan especial de estos problemas, tienen métodos de solución propios.
c) *Problemas de secuenciación*. Este tipo de problemas, aunque son fáciles de formular, resultan bastante difíciles de resolver. Se supone por ejemplo, el caso de un taller que puede efectuar un solo tipo de trabajo a la vez (orden), el que se tiene contratado a entregar en días, a partir de una cierta fecha base, y que además tiene una duración de trabajo de (días y al cual se asocia una multa de pesos por día de retraso

después de los días estipulados. Se supone que el taller recibe órdenes diferentes de trabajo en la fecha base. ¿Cuál debe ser el orden de secuenciación de trabajos que minimice el costo penal total? Sea por ejemplo el siguiente caso.

d) *El problema del agente viajero.* Este problema, concierne a un agente viajero que, saliendo de una determinada ciudad, debe visitar una sola vez ciudades diferentes, y regresar al punto de partida. Si el costo de dirigirse a la ciudad desde la ciudad es , se debe determinar la secuencia de visita de ciudades, tal que el costo total asociado sea mínimo.

Este problema se presentó por primera vez en 1960, pero hay una variedad de métodos que resuelven el problema, dependiendo del tamaño de , el número de ciudades. Una formulación de este problema es el siguiente.

Sean:

Entonces se requiere

Aunque este problema se ilustra en términos de un agente viajero visitando varias ciudades, lo importante es que se trata de una estructura entera, con aplicación a muchos otros problemas, como por ejemplo, la secuenciación de máquinas, herramientas, actividades, rutas, etc. Por ejemplo, una máquina, a la que se le

debe preparar para un trabajo después de haber realizado un trabajo , incurre un costo

e) ***Problema tipo mochila***. Este tipo de problemas de optimización de carácter entero puede darse en dos versiones. En la primera se proporciona un cierto espacio con determinado volumen o capacidad, y éste debe ser llenado con objetos de valor y volumen o capacidad especificados. El problema consiste en llenar ese espacio con el conjunto de objetos más valioso, sin exceder los límites físicos de dicho espacio. La segunda versión consiste en dividir a un objeto en varias porciones de diferente valor. El problema consiste en encontrar la división de mayor valor.

El problema en cuestión se formula como

donde

Cuando es una variable continua (no entera), el problema anterior se convierte en un programa lineal. Este problema puede ser resuelto de diferentes maneras, dependiendo del tamaño de (número de objetos). Como por ejemplo el método de *bifurcación y acotación* que se explicará más adelante.

Los modelos tipos ***mochila*** se usan para resolver problemas de inversiones, problemas de confiabilidad de redes, subrutinas en los métodos de descomposición de programación lineal y problemas de determinación del tamaño de flota de vehículos [22].

A continuación se analizan varios de estos métodos. Al final del análisis se presentará una comparación de los mismos, relativa al tiempo que tardan en resolver un mismo problema *tipo*, en una computadora [22].

2.2 Método de Planos de corte

La familia de métodos llamados de *plano de corte*, fueron los primeros que se utilizaron para abrir la brecha de la *programación entera* y sirvieron para cimentar el camino que más adelante rompería con muchas de las barreras existentes cuando estos métodos fueron publicados. Estos métodos, que tienen su origen con Gomory [14], como métodos duales (factible dual, factible primario-dual) y primarios (factible primario).

La desventaja de los métodos de *planos de corte*, es que resultan muy ineficientes para resolver problemas enteros de tamaño medio. Estos métodos generan en cada iteración una restricción y una variable extra. Sin embargo, su ventaja es que ilustran lo que se pretende hacer con la región de factibilidad del problema entero, para lograr la solución del mismo.

La diferencia entre los métodos duales y primarios en los algoritmos que utilizan *planos de corte*, es la siguiente. Los métodos duales no generan una solución factible del problema, sino hasta que se llega a la solución óptima. En la práctica se ha notado que para ciertos problemas, después de una cantidad bastante grande de iteraciones, lo mejor que se ha logrado, es definir una cota de la función objetivo del problema. Este método dual sin embargo, cuando trabaja, lo hace bien y de prisa. En cambio, los métodos primarios, pueden producir desde la primera iteración una solución factible. El problema estriba en que esta solución factible converge lentísimamente a la solución óptima [22]. Tan lenta es la convergencia,

que el precio comercial en que se vende el tiempo de las computadoras, no conviene económicamente la solución del mismo.

Se entiende por al entero más grande menor que el número es decir

. (2.2.1)

Por ejemplo

Se considera el siguiente problema entero , donde es una matriz de , un vector fila de dimensión , un vector columna de dimensión y un vector columna con entradas enteras de dimensión .

El problema lineal correspondiente a (2.2.2) es

Una restricción típica de (2.2.3) es

donde son las variables no básicas, son las variables básicas y es el valor de la variable básica correspondiente.

Esta restricción puede escribirse como

Poniendo a las partes enteras de un lado de la expresión y a las fracciones del otro, se tiene

Como la parte izquierda de la igualdad es entera, la parte derecha también debe serlo para el problema de programación entera.

Teorema 2.2.1 Dado (2.2.3) se tiene que

Prueba. Dada una restricción típica de (2.2.3):

Se le puede multiplicar por y por , arbitrario, quedando respectivamente

(2.2.4)

(2.2.5)

De la expresión (2.2.4) y (2.2.1) se tiene

(2.2.6)

por lo tanto, como la parte izquierda de la igualdad es entera, se tiene

(2.2.7)

Restando (2.2.7) y (2.2.5) nos queda

o

(2.2.8)

Como es arbitrario, se hace , quedando (2.2.8) como

Y el teorema queda probado.

Los métodos de *planos de corte* están basados fundamentalmente en la restricción

que constituye un *corte.*

Ejemplo 2.2.2 Se deriva un corte de la siguiente restricción

donde es la variable básica. Se tiene, separando la parte entera de la fraccionaria en cada coeficiente

Por lo que el corte resulta ser

2.3 Algoritmo Fraccional de Gomory

El algoritmo fraccional de Gomory consiste en resolver el siguiente problema de programación entera:

La idea es resolver primero el problema de programación lineal asociado y encontrar una solución óptima, elegir una variable básica que no es una solución óptima entera, y luego generar una desigualdad y una variable de holgura en la restricción asociada a esta variable básica con el fin de hacer un corte en la solución de programación lineal, con el método planos de corte descrito anteriormente.

Paso 1. Se resuelve el problema entero como un problema lineal, olvidándose por el momento de las condiciones de integralidad.

Paso 2. Si el resultado óptimo del paso 1 o del paso 3 es entero, pare. Se ha obtenido el resultado óptimo del problema original. De otra manera continúe en el paso 3.

Paso 3. Seleccione el máximo fraccionario y genere un corte

Añádase este corte como una restricción adicional, junto con su variable superflua. Resuélvase el problema por el método dual simplex, cuando se tenga un punto óptimo que no es factible y regrésese al paso 2.

Ejemplo 2.3.3 Resolvamos por el algoritmo fraccionario de Gomory el siguiente problema entero.

Paso 1. La solución del programa lineal genera la siguiente tabla óptima.

Z			
0 0			

Paso 3. Como y no son enteros, se debe generar un corte. El máximo corresponde al primer renglón de la tabla, cuya restricción es.

Por lo tanto, el corte es

o

Agregando el corte y una variable superflua , la tabla queda

Aplicando el *método dual simplex* (2 iteraciones más) se obtiene la siguiente tabla óptima (para el problema lineal), pero que aún no tiene a entero.

Por lo tanto, el nuevo corte se forma de

que tiene el máximo , de todos los posibles candidatos fraccionales. El nuevo corte que se añade es

es decir

La nueva tabla a resolver por medio del *método dual simplex* es

Cuyo resultado óptimo es

Como este resultado óptimo del problema lineal es entero, también es una solución óptima del problema entero.

2.4 Métodos de bifurcación y acotación

El *método de bifurcación y acotación,* es muy elegante y simple, redondea y acota variables enteras, resultantes de la solución de los problemas lineales correspondientes. Este proceso de acotamiento y redondeo se hace de una manera secuencial lógica heurística, que permite eliminar con anticipación un buen número de soluciones factibles alejadas del óptimo a medida que se itera. De tal suerte que si una variable entera está acotada entre un límite inferior entero y un límite superior entero el proceso de bifurcación y acotación sólo analiza un número muy pequeño de todas las posibles soluciones. Para que el lector se dé cuenta de la gran cantidad de posibles soluciones, debe tener presente que sólo la variable puede tomar cualquiera de los siguientes valores enteros: Si se tienen variables enteras, se tendría un número muy grande de posibles combinaciones que se pueden obtener.

2.4.1 Algoritmo de Land-Doig

El algoritmo de *bifurcación y acotación* fue presentado por Land y Doig [17]. El nombre de *bifurcación y acotación* se lo dan posteriormente Little, Murty, Sweeney, Karel [18], a continuación se presenta el *algoritmo de bifurcación y acotación,* suponiendo que se maximiza la función objetivo.

Paso 1. Resuélvase el problema entero por medio del *método simplex* de la programación lineal. Si la solución es entera, pare, se ha conseguido la solución óptima. Si no, continúe en el paso 2.

Paso 2. Escójase arbitrariamente una variable entera cuyo resultado en el paso 1 sea fraccional e igual a .

Paso 3. Resuélvase un par de nuevos problemas, similares al problema anterior, pero uno con la restricción adicional , mientras que el otro tendrá la restricción adicional

Paso 4. De los programas lineales resueltos en el paso 3, inclúyase en el análisis a seguir, sólo aquellos programas cuya solución (entera fraccional) sea mejor a cualquiera de las soluciones enteras conocidas, mayor en el caso de maximización y menor en el caso de minimización.

Paso 5. Selecciónese aquel programa lineal que tenga el máximo valor de la función objetivo. Si las variables enteras tienen valor entero, se ha conseguido la solución óptima. Si no, regrésese al paso 2 con la estructura del problema lineal resuelto en este paso.

Ahora apliquemos el método de ramificación y acotación al siguiente ejemplo.

Ejemplo 2.4.4 Resolvamos

Problema (0).

Iteración 1.

Paso 1. La solución óptima del programa lineal correspondiente es.

Paso 2. Se escoge arbitrariamente y se resuelven dos problemas lineales distintos, uno con la restricción adicional y el otro con la restricción adicional Es decir:

Problema (1)

,

Problema (2)

,

Paso 3. Aplicando el *análisis de sensibilidad,* cuando se agrega una restricción adicional a un problema lineal, o bien el *método de cota superior*, se tienen las siguientes tablas óptimas de los problemas lineales.

Paso 4. Como no hay ninguna solución entera en todo el proceso, se incluyen ambas tablas en el análisis.

Paso 5. Como la mejor función objetivo hasta el momento corresponde a una solución no entera se regresa al paso 2. La estructura seleccionada es la que lleva el número (1).

Iteración 2.

Paso 2. Arbitrariamente, de la estructura (1) se escoge una variable y se resuelven dos nuevos problemas. Uno que es igual al problema (1) más la restricción El otro que es igual al problema (1) más la restricción Es decir:

Problema (3)

,

Paso 3. Aplicando el *método de cota superior,* se obtienen las soluciones óptimas al problema lineal correspondiente a la estructura (3). La estructura (4), no tiene solución factible (el problema es inconsistente) y por lo tanto no se le incluye en el listado de estructuras a analizar.

Paso 4. Por ser una solución entera se incluye en el análisis.

Paso 5. Por ser el mejor valor de la función objetivo y además, ser entero, es la solución óptima, es decir:

O sea

El problema anterior puede describirse por medio de una representación gráfica que está constituida por una red con estructura de árbol, donde cada nodo se le asocia lo siguiente: un número (el de la estructura lineal correspondiente), el valor de las variables y el de la función objetivo, para esa estructura.

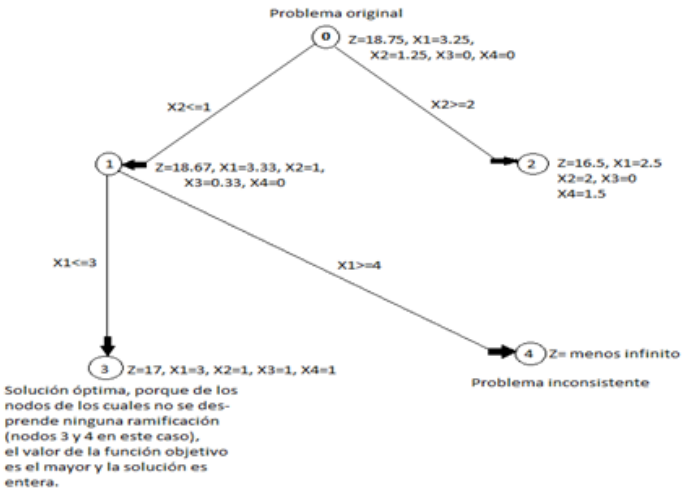

Figura 2.4.1 árbol de soluciones del ejemplo 2.4.4

2.5 Métodos de Enumeración Implícita

Los métodos de *enumeración implícita* son métodos heurísticos, basados en la lógica, que como los métodos de *bifurcación y acotación* resuelven problemas enteros, sin tener que analizar todas las posibles alternativas. Así como el *método de planos de corte* resuelve un problema entero mediante la modificación de la región factible del problema lineal correspondiente y el *método de bifurcación y acotación* mediante la ramificación de problemas que obligan a una variable fraccionada a tomar el valor entero inmediato mayor o menor a la fracción, el método de enumeración implícita resuelve problemas enteros mediante la aceptación o rechazo implícito de ciertas alternativas.

Los *métodos de enumeración implícita* se desarrollaron para resolver problemas del tipo binario, es decir cero-uno. Sin embargo, cualquier problema entero puede convertirse en un problema binario (cero-uno), tomando en cuenta que una variable entera puede representarse por:

donde ó 1, Por ejemplo, se representa por

donde

El método que se explica a continuación resuelve problemas tipo binario (y por ende, cualquier problema entero) y está basado en el trabajo de Balas [5].

2.5.1 Método aditivo de Balas para resolver problemas binarios por enumeración implícita.

El problema a resolver es

Se supone lo siguiente:

a) que la función objetivo se minimiza. En el caso de que sea un proceso de maximización, por las reglas de equivalencia, se convierte el problema en uno de minimización.

b) Se requiere que En caso de que entonces la variable se substituye por otra , donde . Es decir es el complemento de . Así por ejemplo, , quedaría que para el caso de optimización es equivalente al , donde

Con estas dos suposiciones se puede garantizar la existencia de una solución inicial que sea básica factible y dual. Los coeficientes y no necesitan ser números enteros.

A un conjunto específico de variables que describen un problema se le denominará *solución parcial,* ya que de aquellas se generarán las *soluciones completas* del problema. De manera análoga, aquellas variables que no se especifican en la solución parcial, se les llamará *variables libres o variables no especificadas.* Se utiliza la convención en que la componente que se encuentra a la extrema izquierda de una solución parcial, es la primera variable especificada, la segunda a la extrema izquierda es la segunda variable especificada y así con las demás posiciones. Una segunda solución parcial es *continuación* de la primera si todos los elementos de esta última, aparecen en la misma posición que en la primera solución.

Por ejemplo, la solución parcial es una continuación de

Se denota *a una solución parcial cualquiera por:*

cuyo significado es el siguiente:

significa que ha sido seleccionada con un valor uno, de acuerdo con las reglas del método.

significa que ha sido seleccionada con valor cero, de acuerdo con las reglas del método.

significa que , es igual a uno, como consecuencia de la continuación de una solución parcial previa.

significa que es igual a cero, como consecuencia de la continuación de una solución parcial previa.

también puede significar que es uno, debido a que todas las posibles continuaciones de han sido implícitamente analizadas.

también puede significar que es cero debido a que todas las posibles continuaciones de han sido implícitamente analizadas.

Así por ejemplo, la solución parcial significa que se igualó a uno como consecuencia de las reglas del método, y como consecuencia de esa elección se determinó (por deducción o por un análisis implícito *de todas* las consecuencias de tener), que De manera análoga, por tener se determina que y por último, por tener se determina que

Cuando en una solución parcial aparece un elemento sin signo, éste representa a cualquiera de los siguientes elementos:

El *método generalizado de Balas* requiere que todos los costos . Las restricciones de igualdad se trabajan tal cual, pero el resto de las desigualdades deben ser de la forma

Se supone que existe una solución parcial sin signos asociados. Inicialmente se hace o bien igual a , un número positivo muy grande (por ejemplo). Se tienen los siguientes pasos:

Paso 1.

a) Si la solución parcial satisface las restricciones del problema, complete la solución parcial asignando cero a todas las variables libres. Haga igual al valor de la función objetivo correspondiente a la solución parcial. Vaya al paso 5.

b) Si la solución parcial actual no satisface las restricciones del problema, haga igual al valor de la función objetivo correspondiente a la solución parcial y vaya al paso 2.

Paso 2.

Para las variables libres , tales que:

a)

b) por lo menos para una restricción, ,

examine las relaciones (2.5.9)

donde es el conjunto de variables libres que satisfacen *(a) y (b)*

Si ninguna de las restricciones (2.5.9) se viola, entonces no existe una continuación factible. Vaya al paso 5. De otra manera vaya al paso 3.

Paso 3.

a) Si *todas* las relaciones (2.5.9) se satisfacen como una *desigualdad* estricta, determine cuál variable libre del conjunto **,** al tener valor igual a uno, reduce considerablemente la *infactibilidad total.* Por *infactibilidad total* se entiende a la suma, en valor absoluto de las cantidades en las cuales se viola cada restricción. Sea esta variable la Entonces . Se ha maximizado la siguiente expresión :

La nueva solución parcial es Vaya al paso1.

b) Si para alguna restricción (2.5.9), esta se comporta como *una igualdad,* denote por al conjunto de variables libres de esa restricción que tengan Examine la relación

(2.5.10)

Vaya al paso 4.

Paso 4.

a) Si (2.5.10) se satisface, entonces es la única continuación factible que es posible (dada la solución parcial actual). La siguiente solución parcial que se debe analizar es: donde

Regrese al paso 1.

b) Si (2.5.10) no se satisface, entonces no existe una posible continuación de la solución parcial. Vaya al paso 5.

Paso 5.

a) Dada la solución parcial actual encuentre el elemento a la extrema derecha y elimine a todos los elementos que se encuentran a su derecha. Reemplace este elemento (el) por Esta es la nueva solución parcial. Regrese al paso 1.

b) Si no existe un elemento en la solución parcial actual, la enumeración implícita se ha completado y la solución que se tiene en el paso 1 *(a)* es óptima. Si ó , el problema original es inconsistente y por lo tanto no tiene solución.

Ejemplo 2.5.5 Se resuelve por el *método aditivo de Balas* [5], el siguiente problema tipo cero-uno (binario).

Note que este ejemplo ya cumple con las condiciones para toda y las restricciones son de la forma menor o igual. Expresando a las restricciones en la forma se tiene:

Supóngase arbitrariamente una solución
Esto implica que y además:

Por lo que la infactibilidad total es de 3. Para que esta infactibilidad se reduzca es necesario aumentar el valor de ciertas variables libres a uno. Con y todas las demás igual a cero, se tiene:

Por lo que la infactibilidad total es ahora de 4 (no muy buena).

La variable no tiene caso aumentarla a 1, porque perjudicaría aún más a y , ya que de hecho son violadas cuando Si se hace y las demás variables se mantienen en cero, se tiene:

Para una infactibilidad total de 3. Con y las demás variables igual a cero se obtiene:

Para una infactibilidad total de 5. El mismo comentario que se hizo para , es válido para Por lo tanto, lo que más conviene es hacer y se mantiene la primera solución parcial Con las tres restricciones se convierten en:

La solución parcial no es factible, puesto que Para eliminar la violación de la segunda restricción es necesario analizar las posibles continuaciones de Observando a , se concluye que se puede corregir esta violación analizando lo que sucede si aumenta a uno el valor de las variables libres que tengan en este caso y

Con y el resto de las variables libres igual a cero se tiene:

Con y el resto de las variables libres igual a cero se tiene:

Por lo tanto, hacer después de haber hecho da una solución parcial que es factible. El valor de asociada a esta solución parcial factible es Cualquier continuación de aumentaría el valor de y por lo tanto se les puede eliminar. Es decir que *implícitamente se eliminan* las siguientes combinaciones:

Se analiza ahora la solución parcial que significa y que genera:

Como ésta es una solución parcial no factible, *se puede rechazar implícitamente las siguientes continuaciones* de

La solución parcial que se tiene es Se analiza por lo tanto la solución es decir con el resto de las variables cero, lo que origina que:

Como esta solución no es factible *se pueden eliminar implícitamente* todas las soluciones que contengan es decir:

Como ya se han eliminado implícitamente 31 soluciones, y la trigésima segunda es factible y solo hay (32 en este caso) posibles soluciones, se concluye que es óptima [22].

2.6 Comentarios finales de los Métodos de Programación Entera

Se presenta a continuación una indicación relativa de la eficiencia computacional de algunos de los métodos descritos con anterioridad. Existen programas de biblioteca para computadoras digitales que han resuelto por *métodos de enumeración (enumeración implícita y ramificación acotada)* problemas enteros y enteros-mixtos. Algunos resultados de corridas en donde se ha obtenido una solución óptima son:

Tabla 2.6.1

Número de restricciones	*Número de Variables continuas*	*Número de Variables enteras*	*Tiempo de solución En segundos*
288	244	59	27.9
68	0	536	6.6
604	1955	25	1146.0
1244	3884	24	361.6

Benichou [8]: reporta corridas en una computadora **IBM-360** con el mayor problema entero de 69 restricciones y 590 variables enteras, resuelto en 25.7 minutos y con el mayor problema entero mixto de 721 restricciones, 39 variables enteras y 1117 variables continuas en 18 minutos.

Tomlin [25]: reporta haber resuelto en una computadora **UNIVAC-1108,** problemas tipo binario (cero-uno) con 1000 restricciones y 200 variables binarias (cero-uno) en 45 minutos.

Otros resultados publicados de los *métodos de enumeración* (implícita y ramificación acotada) son:

TABLA 2.6.2

Número de restricciones	*Número de Variables enteras*	*Número de Variables continuas*	*Tiempo de solución (minutos)*	*Referencia*
197	100	217	5	Davies, Kendrick y Weitzman [10]
39	14	53	4.5	Aldrich [2]
378	24	136	No se reportó	Aldrich [2]
515	48	851	20	Childress [9]
[531	22 binarias	622	30	Manne [20]

En cuanto a los métodos que utilizan de alguna manera los *planos de corte*, se tienen algunos resultados.

TABLA 2.6.3

Número de restricciones	*Número de variables enteras*	*Tiempo de solución (minutos)*	*Referencia*
60	240	No se reportó	Trauth y Woolsey [26]
37	44	0.5	Srinivasan [23]
91	240	No se reportó	Baugh, Ibaraka Y Muroga [6]

Se hace la aclaración que estos no son algoritmos de *planos de corte* puros, sino que se utilizan en combinación con otros algoritmos, como por ejemplo, *ramificación acotada.* Gorry y Shapiro [13], reportan haber resuelto por medio de métodos basados en *teoría de grupos*, problemas enteros con un promedio de 75 restricciones y 150 variables enteras en menos de un minuto. El problema más grande que reportan haber resuelto es uno con 104 restricciones y 236 variables enteras en 7 minutos, utilizando una computadora **UNIVAC-1108**.

En cuanto a los problemas *tipo mochila* (general no binario), Cabot [42] reporta los siguientes resultados utilizando una computadora **CDC-3600.**

TABLA 2.6.4

Número de objetos en la mochila	*Tiempo promedio de corrida de 50 problemas*
60	3.5 segundos
70	3.5 segundos
80	7.2 segundos

Greenberg y Hegerich [15] reportan haber resuelto con una computadora **IBM 360/67**, problemas tipo *mochila* generales (no binarios), con 5000 objetos diferentes *(n=5000),* en aproximadamente 4 minutos.

Existe una amplia variedad de métodos para resolver problemas enteros y aun cuando se ha alcanzado cierto éxito, el tamaño de los problemas enteros a resolver sigue siendo un obstáculo. Los métodos más populares son los de *bifurcación y acotación.* Si un problema tiene menos de 50 variables enteras, seguramente puede ser resuelto por cualquiera de los programas comerciales de biblioteca, que tienen disponibles las compañías de computadoras. Los problemas con un número de variables enteras que oscilan entre 50 y 100, requieren de programas más especiales, que no abundan, como los primeros. Para más de 100 variables enteras, se requieren técnicas muy especializadas (sobre todo para usar programas de programación lineal como subrutinas), que hacen que los buenos programas comerciales escaseen o no existan. En estos últimos casos, es necesario, construir programas propios, tarea que no es nada sencilla.

Entre los programas comerciales de computadoras digitales, que pudieran utilizarse para resolver problemas enteros (con no más de 100 variables enteras) están:

Ophelie para la **CDC-6600,** *Extendended MPS* para la **IBM 360/370** *ophelie, extendended y Scientific Control Systems Umpire* para la **UNIVAC-1108.**

CAPÍTULO 3 HEURÍSTICAS

Dada la dificultad práctica para resolver de forma exacta (simplex, ramificación y acotación, etc.) toda una serie de importantes problemas combinatorios para los cuales, por otra parte, es necesario ofrecer alguna solución dado su interés práctico, comenzaron a aparecer algoritmos que proporcionan soluciones factibles (es decir, que satisfacen las restricciones del problema), las cuales, aunque no optimicen la función objetivo, se supone que al menos se acercan al valor óptimo en un tiempo de cálculo razonable. Podríamos llamarlas en lugar de óptimas, "satisfactorias".

Este tipo de algoritmos se denominan *heurísticas,* que proviene del griego *heuriskein* (encontrar) que siendo más exactos lo que hacen es buscar. Aunque en un principio no fueron bien vistas en los círculos académicos acusadas de escaso rigor matemático, su interés práctico como herramienta útil que da soluciones a problemas reales, les fue abriendo poco a poco las puertas, sobre todo a partir de los años setenta con la proliferación de resultados en el campo de la complejidad computacional. En este capítulo se presenta un resumen acerca de estos métodos de solución tomada de [12].

Una posible manera de definir estos métodos es como *"procedimientos simples, a menudo basados en el sentido común, que se supone ofrecerán una buena solución (aunque no necesariamente la óptima) a problemas difíciles, de un modo fácil y rápido".*

Son varios los factores que pueden hacer interesante la utilización de algoritmos heurísticos para la solución del problema:

a) *Cuando no existe un método exacto de resolución o éste requiere mucho tiempo de cálculo.* Ofrecer entonces una solución que sólo sea aceptablemente buena resulta de interés frente a la alternativa de no tener ninguna solución en absoluto.

b) *Cuando no se necesita la solución óptima.* Si los valores que adquiere la función objetivo son relativamente pequeños, puede no merecer la pena esforzarse en hallar una solución óptima que, por otra parte, no representará un beneficio importante respecto a una que sea simplemente sub-óptima. En este sentido, si puede ofrecer una solución mejor que la actualmente disponible, esto puede ser ya de interés suficiente en muchos casos.
c) *Cuando los datos son poco factibles.* En este caso, o bien cuando el modelo es una simplificación de la realidad puede carecer de interés buscar una solución exacta, dado que de por sí ésta no será más que una aproximación de la real, al basarse en datos que no son reales.
d) *Cuando hay limitaciones de tiempo, espacio* (para almacenamiento de datos), etc., obliguen al empleo de métodos de rápida respuesta, aun a costa de la precisión.
e) *Como paso intermedio en la aplicación de otro algoritmo.* A veces son usadas soluciones heurísticas como punto de partida de algoritmos exactos de tipo iterativo.

Una importante ventaja que presentan las heurísticas respecto a las técnicas que buscan soluciones exactas es que, por lo general, permiten una mayor flexibilidad para el manejo de las características del problema. No suele resultar complejo diseñar algoritmos heurísticos. Además, generalmente ofrecen más de una solución, lo cual permite ampliar las posibilidades de elección del que decide, sobre todo cuando existen factores no cuantificables que no han podido ser añadidos en el modelo, pero que también deben ser considerados.

Por otra parte, suele ser más fácil de entender la fundamentación de las heurísticas que los complejos métodos matemáticos que utilizan la mayoría de técnicas exactas.

Una de las mejores descripciones de la importancia de las heurísticas es la resolución de problemas interesantes. Si bien esta tesis se centra en la solución del problema de la mochila, muchas técnicas heurísticas tienen una aplicabilidad más general. Las páginas siguientes son algo concisas, pero tan sencillas como

fueron posibles, y se basan en un estudio concreto y serio de los métodos heurísticos de solución. Este tipo de estudio, denominado heurístico, no es una moda actual sino que tiene un largo pasado y, quizás, algún futuro.

Existen dos formas fundamentales de incorporación de conocimiento heurístico específico del dominio a un proceso de búsqueda basado en reglas:

- En las mismas reglas. Por ejemplo, las reglas de un sistema para jugar ajedrez podrían no solo describir el conjunto de movimientos legales, sino incluso un conjunto de movimientos “sensatos” determinados por el programador de las reglas.
- Como una función heurística que evalúa los estados individuales del problema y determina su grado de “deseabilidad”.

Una *función heurística* es una correspondencia entre las descripciones de estados del problema hacia alguna medida deseable, normalmente representada por números. Una función heurística bien diseñada puede desempeñar un importante papel de guía eficiente del proceso de búsqueda hacia una solución. En ocasiones, una función heurística muy sencilla puede proporcionar buenas estimaciones sobre si una ruta es adecuada o no. En otras situaciones, deben emplearse funciones heurísticas más complejas.

La forma en que se implementa la función, en general, no tiene importancia. El programa que utilice los valores de la función tendrá que minimizar o maximizar según sea adecuado. El propósito de una función heurística es el de guiar el proceso de búsqueda en la dirección más provechosa sugiriendo qué camino se debe seguir primero cuando hay más de uno disponible. Cuanto más exactamente estime la función heurística los méritos de cada nodo del árbol o grafo, más directo será el proceso de solución.

Después de todo, sería posible hacer una función heurística perfecta realizando una búsqueda completa desde el nodo en cuestión, determinando si conduce a una solución adecuada. En general, existe un compromiso entre el costo de

evaluación de una función heurística y el ahorro de tiempo de búsqueda que proporciona la función.

Algunas heurísticas se usarán para definir la estructura de control que guía la aplicación de las reglas en el proceso de búsqueda. Otras, tal y como se verá, se incorporan en las propias reglas. En ambos casos, representan el conocimiento tanto general como específico del mundo que hace que sea abordable solucionar problemas complejos [12].

3.1 Tipos de Heurísticas

Existen diferentes tipos de heurísticas, según el modo en que se buscan y construyen las soluciones. Una posible clasificación es la siguiente:

a) *Métodos constructivos.* Consisten en ir paulatinamente añadiendo componentes individuales a la solución hasta que se obtiene una solución factible. El más popular de estos métodos lo constituyen los algoritmos *golosos o devoradores (greedy),* los cuales construyen paso a paso la solución buscando el máximo beneficio en cada paso.
b) *Métodos de descomposición.* Se trata de dividir el problema en sub-problemas más pequeños, siendo el output de uno el input de su siguiente, de forma que al resolverlos todos obtengamos una solución para el problema global (divide y vencerás). Un ejemplo de aplicación de este método a un problema de programación lineal mixta, consiste en decidir de alguna forma (quizás mediante otra heurística) una solución para las variables enteras, y luego resolver el problema de programación lineal obtenido al sustituir esas variables por su valor. Algunos autores diferencian entre métodos de descomposición (los subproblemas se resuelven en cascada) y métodos de partición (cuando los subproblemas son independientes entre sí).
c) *Métodos de reducción.* Tratan de identificar alguna característica que presumiblemente deba poseer la solución óptima y de ese modo simplificar el problema. Así, puede detectarse que alguna variable deba tomar siempre el valor cero, que otras están correlacionadas, etc.

d) *Manipulación del modelo.* Estas heurísticas modifican la estructura del modelo con el fin de hacerlo más sencillo de resolver, deduciendo, a partir de su solución, la solución del problema original. Pueden consistir en reducir el espacio de soluciones, o incluso aumentarlo.
e) *Métodos de búsqueda por entornos.* Dentro de esta última categoría es donde se encuentra la mayoría de las heurísticas. Estos métodos parten de una solución factible inicial (obtenida quizá mediante otra heurística) y, mediante alteraciones de esa solución, van pasando de forma iterativa, y mientras no se cumpla un determinado criterio de parada, a otras factibles de su "entorno", almacenando como óptima la mejor de las soluciones visitadas.

Un concepto clave en ellas es cómo realizar el paso de una solución factible a otra. Para ello resulta de interés definir lo que es el *entorno* de la solución , es decir, el conjunto de soluciones "parecidas" a ella. El significado aquí de "parecido" debe entenderse como la posibilidad de obtener una solución a partir de la realizando sólo una operación elemental, llamada movimiento, sobre (eliminar o añadir un elemento en el subconjunto solución, intercambiar elementos en la permutación , etc.).

Estos métodos se basan, por tanto, en buscar de entre los elementos del entorno de la solución actual, aquél que tenga un mejor valor de acuerdo con algún criterio predefinido, moverse a él y repetir la operación hasta que se considere que no es posible hallar una mejor solución, bien porque no hay ningún elemento en el entorno de la solución actual, o bien porque se verifique algún criterio de parada.

Una clase especial dentro de este método lo constituye el método de "búsqueda local o de descenso". En él, en cada iteración el movimiento se produce desde la solución actual a una de su entorno que sea mejor que ella, finalizando la búsqueda cuando todos los de su entorno sean peores. Es decir, la solución final será siempre un óptimo local.

Uno de los mayores inconvenientes con los que se enfrentan estas técnicas es la existencia de óptimos locales que no sean absolutos. Una solución se dice que es “óptimo local” si entonces (suponiendo que el objetivo es minimizar y que representa el costo de la solución). Si a lo largo de la búsqueda se cae en un óptimo local, en principio la heurística no sabría continuar pues se quedaría “pegada” en ese punto. Una primera posibilidad para salvar esa dificultad consiste reiniciar la búsqueda desde otra solución inicial y confiar en que, es esta ocasión, la exploración siga por otros caminos. Una de las técnicas que veremos a continuación (*recocido simulado, búsqueda tabú*) que siguen diferentes estrategias para evitar caer en óptimos locales: la primera manteniendo una memoria de la ruta seguida para identificarlos, y la segunda permitiendo con cierta probabilidad la aceptación de caer en óptimos locales.

Otro de los problemas de estas estrategias heurísticas es el conseguir hacerse independientes de la solución inicial de la que se parta. Es obvio que el punto inicial tiene una gran influencia sobre la posibilidad de caer o no en un óptimo local. A pesar de que los métodos heurísticos sólo evalúan normalmente un número pequeño de alternativas del número total posible. Una heurística bien diseñada puede aprovechar esta propiedad y explorar exclusivamente las soluciones más interesantes [12].

3.2 Recocido Simulado

El recocido simulado al comienzo del proceso, puede realizar algunos movimientos descendentes. La idea consiste en realizar una exploración lo suficientemente amplia al principio, ya que la solución final es relativamente insensible con el estado inicial. Así, se disminuye la probabilidad de caer en un máximo local, una meseta o cresta.

El enfriamiento simulado Kirkpatrick [12] visto como proceso computacional, se basa en el proceso físico de la aleación, en la que ciertas sustancias físicas como los metales se funden (es decir incrementan sus niveles de energía) para luego sufrir un proceso gradual de enfriamiento hacia un estado sólido. El objetivo de este proceso es alcanzar un estado final de mínima energía. De esta forma, el

proceso consiste en el descenso por un valle en el que la función objetivo es el nivel de energía. Las sustancias físicas normalmente evolucionan hacia configuraciones de baja energía, de forma que el descenso ocurre de forma natural. Sin embargo, existe cierta probabilidad de que se produzcan transiciones hacia estados con energía más alta. Esta probabilidad viene dada por la función:

donde es el cambio positivo en el nivel de energía, es la temperatura, y es la constante de Boltzmann. Así, durante el descenso del valle que ocurre durante el enfriamiento, la probabilidad de que ocurran grandes saltos positivos es menor que la probabilidad de que ocurra uno más pequeño. También, la probabilidad de que se produzca un salto positivo decrece conforme la temperatura va bajando. De esta forma, estos saltos son más probables en los comienzos del proceso, cuando la temperatura baja. Una forma de caracterizar este proceso es diciendo que se permiten los movimientos descendentes en cualquier momento. Los grandes saltos ascendentes sólo se permiten al principio, pero conforme el proceso va progresando, sólo se permitirán movimientos ascendentes relativamente pequeños que finalmente el proceso converja a una configuración de mínimo local.

La velocidad con la que se enfría el sistema a lo largo del proceso se denomina *programa de enfriamiento*. Los procesos físicos de enfriamiento se producen con demasiada rapidez, se formarán regiones estables de alta energía. En otras palabras, se alcanza un mínimo local pero no global. Si, por el contrario, se utiliza un programa más lento, es más probable que se forme una estructura cristalina uniforme, que corresponde a un mínimo global. Pero, si el programa es demasiado lento, se desaprovecha el tiempo. A altas temperaturas, en las que se permiten movimientos básicamente aleatorios, no ocurre nada de provecho. A bajas temperaturas se puede malgastar mucho tiempo después de que se haya formado la estructura final. El plan de enfriamiento óptimo para cada problema de enfriamiento específico se suele hallar de forma empírica [12].

Todas estas propiedades del enfriamiento físico se pueden utilizar para definir un proceso análogo de enfriamiento simulado, que podría aplicarse (aunque no siempre con efectividad) en el momento en el que sea posible utilizar una escalada simple. Para este proceso análogo, sufre una generalización, de forma que en vez de representar un cambio en la energía, representa un cambio en la función objetivo, cualquiera que ésta sea. La analogía con es un poco más complicada. En el proceso físico, la temperatura es una magnitud bien definida, medida en unidades estándar. La variable describe la correspondencia que existe entre las unidades de temperatura y las unidades de energía. Como en el proceso análogo, las unidades de y son artificiales, tiene sentido incorporar a , de forma que se seleccionen valores de que hagan que el algoritmo se comporte adecuadamente. De esta forma, revisamos la formula probabilística:

Sin embrago, es necesario elegir un plan para los valores de (que todavía denominamos temperatura). Después de la representación del algoritmo de recocido simulado, se explicara brevemente esta situación.

3.2.1 Algoritmo: recocido simulado

1. Evaluar el estado inicial. Si también es el estado objetivo, devolverlo y terminar. En caso contrario, continuar con el estado inicial como el estado actual.
2. Inicializar la variable EL-MEJOR-HASTA-AHORA con el estado actual.
3. Inicializar de acuerdo con el programa de enfriamiento.
4. Repetir hasta que se encuentre una solución o hasta que no queden operadores que aplicar al estado actual.
   (a) Seleccionar un operador que aún no se haya aplicado al estado actual para producir un estado nuevo.
   (b) Evaluar el nuevo estado. Calcular: =(valor del estado actual)-(valor del nuevo estado)
   - Si el nuevo estado es un estado objetivo, devolverlo y terminar.

- Si no es un estado objetivo, pero es mejor que el estado actual, convertirlo en el estado actual. Hacer también que EL-MEJOR-HASTA-AHORA sea el nuevo estado.
- Si no es mejor que el estado actual, convertirlo en el estado actual con la probabilidad definida anteriormente. Este paso normalmente se implementa invocando a un generador de números aleatorios para que genere un número de rango Sólo si este número es menor que , se acepta el movimiento.

(c) Revisar cuando sea necesario, de acuerdo con el programa de enfriamiento.

5. Devolver como respuesta EL-MEJOR-HASTA-AHORA.

Para la implementación de este algoritmo se necesita elegir un programa de enfriamiento que se compone de tres partes. La primera de éllas es el valor inicial de la temperatura. La segunda es el criterio que se sigue para decidir cuándo se va a reducir la temperatura del sistema. La tercera es la magnitud del decremento que sufre la temperatura en cada cambio. Existe un cuarto componente en el programa de enfriamiento: cuando terminar. El recocido simulado se utiliza frecuentemente en aquellos problemas en los que el número de movimientos que pueden realizarse en un cierto estado, es muy elevado. Con esta clase de problemas, es mejor usar algún criterio que incorpore el número de movimientos que se han intentado a partir del hallazgo de una mejora.

Los experimentos que se han realizado con el recocido simulado aplicándolos a varios problemas, sugieren que la mejor forma de encontrar un programa de enfriamiento se logra intentando varios y observando tanto la calidad de la solución como la forma en la que converge el proceso. Para empezar, hay que tener en cuenta que conforme se aproxima a cero, la probabilidad de que se acepten movimientos hacia estados peores se acerca también a cero, y el recocido simulado es, por lo tanto, idéntico a una escalada simple. El segundo aspecto a tener en cuenta es que lo que realmente ocurre en el cálculo de

probabilidad de aceptar un movimiento, es la razón De esta forma, es importante que los valores de estén escalados de forma que esta razón sea significativa. Por ejemplo, podría inicializarse con un determinado valor, de forma que, para un valor normal de , fuera 0.5 [11].

3.3 Backtracking (método de vuelta atrás)

Dentro de las técnicas de diseño de algoritmos, el método de vuelta atrás (del inglés Backtracking), fue acuñado por primera vez por el matemático estadounidense D.H Lehmer en la década de 1950. Es uno de los algoritmos de más amplia utilización, en el sentido de que puede aplicarse en la resolución de un gran número de problemas, especialmente los de optimización.

El método vuelta atrás proporciona una manera sistemática de generar todas las posibles soluciones siempre que dichas soluciones sean susceptibles de resolverse en etapas.

En su forma básica la vuelta atrás se asemeja a un recorrido de profundidad dentro de un árbol cuya existencia solo es implícita, y que denominaremos *árbol de expansión.* Este árbol es conceptual y solo haremos uso de su organización como tal, en donde cada nodo de nivel representa una parte de la solución y está formado por etapas que se suponen ya realizadas. Sus hijos son las prolongaciones posibles al añadir una nueva etapa. Para examinar el conjunto de posibles soluciones es suficiente recorrer este árbol construyendo soluciones parciales a medida que se avanza en el recorrido.

En este recorrido pueden suceder dos cosas. La primera es que tenga éxito si, procediendo de esta manera, se llega a una solución (una hoja del árbol). Si lo único que buscamos era una solución al problema, el algoritmo finaliza aquí; ahora bien, si lo que buscábamos eran todas las soluciones o la mejor de entre todas ellas, el algoritmo seguirá explorando el árbol en búsqueda de soluciones alternativas.

Por otra parte, el recorrido no tiene éxito si en alguna etapa la solución parcial construida hasta el momento no se puede completar; nos encontramos en lo que

llamamos *nodos fracaso.* En tal caso, el algoritmo vuelve atrás (y de ahí su nombre) en su recorrido eliminando los elementos que se hubieran añadido en cada etapa a partir de ese nodo. En este retroceso, si existe uno o más caminos aun no explorados que puedan conducir a la solución, el recorrido del árbol continúa por ellos.

La filosofía de estos algoritmos no sigue unas reglas fijas en la búsqueda de las soluciones. Podríamos hablar de un proceso de prueba y error en el cual se va trabajando por etapas construyendo gradualmente una solución. Para muchos problemas esta prueba en cada etapa crece de una manera exponencial, lo cual es necesario evitar.

Gran parte de la eficiencia (siempre relativa) de un algoritmo de vuelta atrás proviene de considerar al menor conjunto de nodos que puedan llegar a ser soluciones, aunque siempre asegurándonos de que el árbol "podado" siga conteniendo todas las soluciones. Por otra parte debemos tener cuidado a la hora de decidir el tipo de restricciones que comprobamos en cada nodo a fin de detectar nodos fracaso. Evidentemente el análisis de estas restricciones permite ahorrar tiempo, al delimitar el tamaño del árbol a explorar. Sin embrago esta evaluación requiere a su vez tiempo extra, de manera que aquellas restricciones que vayan a detectar pocos nodos fracaso no serán normalmente interesantes. No obstante, podríamos decir que las restricciones sencillas son siempre apropiadas, mientras que las más sofisticadas que requieren más tiempo en su cálculo deberían reservarse para situaciones en las que el árbol que se genera sea muy grande.

Vamos a ver cómo se lleva a cabo la búsqueda de soluciones trabajando sobre este árbol y su recorrido. En líneas generales, un problema puede resolverse con un algoritmo vuelta atrás cuando la solución puede expresarse como una

donde cada una de las componentes de este vector es elegida en cada etapa de entre un conjunto finito de valores. Cada etapa representará un nivel en el árbol de expansión.

En primer lugar debemos fijar la descomposición en etapas que vamos a realizar y definir, dependiendo del problema, la que representa la solución del problema y el significado de sus componentes . Una vez que veamos las posibles opciones de cada etapa quedará definida la estructura del árbol a recorrer. Vamos a ver a través de un ejemplo cómo es posible definir la estructura de árbol de expansión [4].

Ejemplo: La mochila

El problema de la mochila descrito anteriormente, nos planteamos aquí dar una solución al problema utilizando vuelta atrás.

Recordemos el enunciado del problema. Dados objetos con pesos y beneficios , y dada una mochila capaz de albergar hasta un máximo peso (capacidad de la mochila), queremos encontrar cuales de los objetos hemos de introducir en la mochila de forma que la suma de los beneficios de los objetos escogidos sea máxima, sujeto a la restricción de que tales elementos no pueden superar la capacidad de la mochila [4].

Solución

Este es uno de los problemas cuya resolución más sencilla se obtiene utilizando la técnica de vuelta atrás, puesto que basta expresar la solución al problema como una de valores donde tomará los valores 1 o 0 dependiendo de que el objeto sea introducido o no. El árbol de expansión resultante es, por tanto trivial.

Sin embargo, y puesto que se trata de un problema de optimización, podemos aplicar una poda para eliminar aquellos nodos que no conduzcan a una solución óptima. Para ello utilizaremos una función (*cota*) que describiremos más adelante.

Como su versión recursiva no plantea mayores dificultades por tratarse de un ejercicio de aplicación, este problema lo resolveremos mediante un algoritmo iterativo programado en MATLAB.

```
CONST n =…; (* número de elementos *)

M =…; (* capacidad de la mochila *)

TYPE REGISTRO = RECORD peso, beneficio: REAL END;

ELEMENTOS = ARRAY [1...n] OF REGISTRO;

MOCHILA = ARRAY [1…n] OF CARDINAL;

PROCEDURE Mochila (elem: ELEMENTOS; capacidad: REAL; VAR X: MOCHILA; VAR
peso_final, beneficio_final: REAL);

VAR peso_en_curso, beneficio_en_curso: REAL;

sol : MOCHILA; (* solución en curso *)

k : CARDINAL;

BEGIN

peso_en_curso: = 0.0;

beneficio_en_curso: = 0.0;

beneficio_final: = -1.0;

k: = 1;

LOOP

WHILE (k<=n) AND (peso_en_curso+elem[k].peso<=capacidad) DO

peso_en_curso: = peso_en_curso+elem[k].peso;

beneficio_en_curso: = beneficio_en_curso+elem[k].beneficio;

sol [k]: = 1;

INC (k)

END;

IF k>n THEN

beneficio_final: = beneficio_en_curso;

peso_final: = peso_en_curso;

k: = n;

X: = sol;

ELSE
```

```
Sol [k]: = 0
END;
WHILE Cota (elem, beneficio_en_curso, peso_en_curso, k, capacidad)
<=beneficio_final DO
WHILE (k<>0) AND (sol [k] <>1) DO DEC (k) END;
IF k=0 THEN EXIT END;
sol [k] : = 0;
peso_en_curso: = peso_en_curso-elem[k].peso;
beneficio_en_curso: = beneficio_en_curso-elem[k].beneficio
END;
INC (k)
END
END Mochila;
```

La función *Cota* es la que va a permitir realizar la poda del árbol de expansión para aquellos nodos que no lleven a la solución óptima. Para ello vamos a considerar que los elementos iniciales están todos ordenados de forma decreciente por su radio beneficio/peso. De esta forma, supongamos que nos encontramos en el paso *k*-ésimo, y que disponemos de un beneficio acumulado . Por la manera en como hemos ido construyendo el vector sabemos que

Para calcular el valor máximo que podríamos alcanzar con ese nodo , vamos a suponer que llenamos el resto de la mochila con el mejor de los objetos que nos quedan por analizar. Como los tenemos dispuestos en orden decreciente de radio beneficio/peso, este mejor objeto será el siguiente . Este valor, aunque no tiene por qué ser alcanzable, nos permite dar una cota superior del valor que podemos “aspirar” si seguimos por esa rama del árbol:

Con esto:

```
PROCEDURE Cota (e: ELEMENTOS; b: REAL; p: REAL; k: CARDINAL; cap: CARDINAL): REAL;

VAR i: CARDINAL; ben_ac, peso_ac: REAL; (* acumulados *)

BEGIN

ben_ac: = b; peso_ac: =p;

FOR I: = k+1 TO n DO

Peso_ac: = peso_ac+e[i].beneficio

IF peso_ac < cap THEN ben_ac: = ben_ac+e[i].beneficio

ELSE

RETURN (ben_ac+(1.0-(peso_ac-cap)/e[i].peso)*(e[i].beneficio))

END

END;

RETURN (beneficio_acumulado)

END Cota;
```

El tipo de poda que hemos realizado al árbol de expansión está más cerca de las técnicas de poda que se utilizan en los algoritmos de ramificación y poda que de las típicas restricciones definidas para los algoritmos vuelta atrás.

Aparte de las diferencias existentes entre ambos tipos de algoritmos en cuanto a la forma de recorrer el árbol (mucho más flexible en la técnica de ramificación y poda) y la utilización de estructuras globales en vuelta atrás (frente a los nodos “autónomos” de la ramificación y poda), no existe una frontera clara entre los procesos de poda de unos y otros. En cualquier caso, repetimos la importancia de la poda, esencial para convertir en tratables estos algoritmos de orden exponencial [4].

3.4 Hill climbing (escalada)

En el método existe realimentación a partir del procedimiento de prueba que se usa para ayudar al generador a decidirse por cual dirección debe moverse en el espacio de búsqueda. En un procedimiento de generación y prueba puro, la función de prueba responde sólo un sí o no. La función de prueba se amplía

mediante una función heurística o función objetivo que proporcione una estimación de lo cercano que se encuentra un estado al estado objetivo.

El método de la escalada se utiliza frecuentemente cuando se dispone de una buena función heurística para evaluar los estados, pero cuando no se dispone de otro tipo de conocimiento provechoso.

Una forma de caracterizar los problemas es respondiendo a la siguiente pregunta, “una solución adecuada ¿es absoluta o relativa?”. Las soluciones son absolutas cuando es posible reconocer un estado objetivo simplemente con examinarlo. Las soluciones relativas únicamente existen en problemas de maximización o minimización, tales como el problema de la mochila. En estos problemas, no existe un estado objetivo a priori. Para problemas como éstos, parece lógico abandonar la escalada si no existe un estado razonable alternativo al que moverse [11].
La forma más sencilla de implementar el método de la escalada es la siguiente:

3.4.1 Algoritmo: escalada simple

1. Evaluar el estado inicial. Si también es el estado objetivo, devolverlo y terminar. En caso contrario, continuar con el estado inicial como estado actual.
2. Repetir hasta que se encuentre una solución o hasta que no queden nuevos operadores que aplicar al estado actual:
    a) Seleccionar un operador que no haya sido aplicado con anterioridad al estado actual y aplicarlo para generar un nuevo estado.
    b) Evaluar el nuevo estado.
        i. Si es un estado objetivo, devolverlo y terminar.
        ii. Si no es un estado objetivo, pero es mejor que el estado actual, convertirlo en el estado actual.
        iii. Si no es mejor que el estado actual, continuar con el bucle.

El funcionamiento de este algoritmo, consiste en el uso de una función de evaluación como una forma de introducir conocimiento específico de la tarea realizada en el proceso de control. Este procedimiento es lo que hace a éste y a otros algoritmos de búsqueda heurística, que tengan la capacidad de resolver problemas complejos.

Nótese que en este algoritmo se ha formulado la pregunta, "¿es un estado mejor que otro?". Para que el algoritmo pueda funcionar, es necesario proporcionar una definición precisa de término mejor. En algunos casos, significa un valor más alto de una función heurística; en otros, significa un valor más bajo.
No importa lo que signifique siempre que a lo largo de una escalada específica sea consistente con su interpretación [11].

3.4.2 Escalada por la máxima pendiente

Una variación útil del método de escalada simple consiste en considerar todos los posibles movimientos a partir del estado actual y elegir al mejor de ellos como nuevo estado. Este método se denomina método de *escalada por la máxima pendiente (steep-ascent Hill climbing) o búsqueda del gradiente (gradient search).* Nótese el contraste con el método básico, en el que el primer estado que aparezca que sea mejor que el actual se selecciona como el estado actual. El algoritmo funciona así [11].

3.4.3 Algoritmo: escalada por la máxima pendiente

1. Evaluar el estado inicial. Si también es el estado objetivo, devolverlo y terminar. En caso contrario, continuar con el estado inicial como estado actual.
2. Repetir hasta que se encuentre una solución o hasta que una iteración completa no produzca un cambio en el estado actual:
    a) Sea SUCC un estado tal que algún posible sucesor del estado actual sea mejor que este SUCC.
    b) Para cada operador aplicado al estado actual hacer lo siguiente:
        i. Aplicar el operador y generar un nuevo estado.

ii. Evaluar el nuevo estado. Si es un estado objetivo, devolverlo y terminar. Si no, compararlo con SUCC. Si es mejor, asignar a SUCC este nuevo estado. Si no es mejor, dejar SUCC como está.

c) Si SUCC es mejor que el estado actual, hacer que el estado actual sea SUCC.

Tanto la escalada básica como la de máxima pendiente pueden no encontrar una solución. Cualquiera de los dos algoritmos puede acabar sin encontrar un estado objetivo, y en cambio encontrar un estado del que no sea posible generar nuevos estados mejores que él. Esto ocurre si el programa se topa con un máximo local, una meseta o una cresta.

Un *máximo local* es un estado que es mejor que todos sus vecinos, pero que no es mejor que otros estados de otros lugares. En un máximo local, todos los movimientos producen estados peores. Los máximos locales son particularmente frustrantes porque frecuentemente aparecen en las cercanías de una solución. En este caso se denominan *estribaciones (foot-hills).*

Una *meseta (plateu)* es un área plana del espacio de búsqueda en la que un conjunto de estados vecinos posee el mismo valor. En una meseta no es posible determinar la mejor dirección a la que moverse haciendo comparaciones locales.

Una *cresta (ridge)* es un tipo especial de máximo local. Es un área del espacio de búsqueda más alta que las áreas circundantes y que además posee en ella misma una inclinación (la cual se podría escalar). Pero la orientación de esta región alta, comparada con el conjunto de movimientos disponibles y direcciones en la que moverse, hace que sea imposible atravesar la cresta mediante movimientos simples.

Existen algunas formas de evitar estos problemas, si bien estos métodos no dan garantías:

- Volver atrás hacia algún nodo anterior e intentar un camino diferente. Es especialmente razonable si el nodo posee otra dirección que dé la impresión de ser tan prometedora o casi tan prometedora, como la que se eligió. Para implementar esta estrategia, se debe mantener una lista de caminos que casi se han seguido y volver a uno de ellos, si el camino que se ha seguido da la impresión de ser un callejón sin salida. Este método es adecuado para superar máximos locales.
- Realizar un gran salto en alguna dirección para intentar buscar en una nueva parte del espacio de búsqueda. Este método está especialmente indicado para superar mesetas. Si la única regla aplicable describe pequeños pasos, aplicarla varias veces en la misma dirección.
- Aplicar dos o más reglas antes de realizar la evaluación. Esto se corresponde con movimientos en varias direcciones a la vez. Este método es especialmente bueno para superar las crestas.

Incluso con estas tres medidas, la escalada no es siempre muy eficaz. Especialmente es inadecuada para problemas en los que el valor de la función heurística cambia bruscamente al alejarse de una solución. Esto ocurre frecuentemente cuando aparece algún tipo de efecto umbral. La escalada es un método local, lo que significa que decide cuál va a ser el siguiente movimiento atendiendo únicamente a las consecuencias “inmediatas” que va a tener esa elección, en lugar de explorar exhaustivamente todas las consecuencias. El método de escalada comparte con otros métodos locales, la ventaja de provocar una explosión combinatoria menor que los métodos globales. Sin embargo también comparte con otros métodos locales una falta de garantías de que va a resultar eficaz. Aunque es verdad que el procedimiento de escalada en sí mismo solo realiza un movimiento hacia delante y nunca otros más alejados, esta exploración puede, de hecho, utilizar una cantidad arbitraria de información global si esta información está codificada en la información heurística.

Desafortunadamente, no siempre es posible construir una función heurística perfecta. Y si es posible disponer del conocimiento perfecto, podría no ser manipulable computacionalmente. Imagine una función heurística que calcule el valor de un estado invocando su propio procedimiento de resolución del problema para mirar alrededor del estado dado para encontrar la solución. En este caso se sabe el costo exacto para encontrar una solución y puede devolverse este costo como su valor. Una función heurística como ésta, convierte el procedimiento local de escalada en un método global introduciendo un método global dentro de él. Sin embargo se pierden las ventajas computacionales de los métodos locales. De esta forma sigue siendo verdad que la escalda puede resultar muy ineficiente con problema grandes y escabrosos. Sin embargo, suele ser adecuado si se combina con otros métodos que consigan que se muevan correctamente por la vecindad en general [11].

3.5 Búsqueda Tabú

La búsqueda Tabú surge, en un intento de dotar de “inteligencia” a los algoritmos de búsqueda local. Según Fred Glover [24], su primer definidor, “la búsqueda tabú guía un procedimiento de búsqueda local para explorar el espacio de soluciones más allá del óptimo local”. La búsqueda tabú toma de la Inteligencia Artificial el concepto de memoria y lo implementa mediante estructuras simples con el objetivo de dirigir la búsqueda teniendo en cuenta la historia de ésta, es decir, el procedimiento trata de extraer información de lo sucedido y actuar en consecuencia.

En este sentido puede decirse que hay un cierto aprendizaje y que la búsqueda es inteligente. La búsqueda tabú permite moverse a una solución aunque no sea tan buena como la actual, de modo que se pueda escapar de óptimos locales y continuar estratégicamente la búsqueda de soluciones aún mejores.

Se presentan primero algunos conceptos propios de los métodos de búsqueda y luego los conceptos específicos de la búsqueda tabú [24].

3.5.1 Conceptos de los métodos de búsqueda

- Dado un problema P de optimización combinatoria utilizaremos x para denotar el conjunto de soluciones posibles del problema y c para la función objetivo.
- Cada solución tiene un conjunto de posibles soluciones asociadas, que se denomina entorno o vecindario de x y se denota como N(x).
- Cada solución del entorno puede obtenerse directamente a partir de x mediante una operación llamada movimiento, que consiste en una búsqueda local que se desplaza por vecindades para moverse iterativamente desde una solución hacia a una solución .
- La solución Inicial dependerá del algoritmo específico que la genera, se puede usar una heurística para proporcionar una buena solución inicial o se puede partir de una solución obtenida aleatoriamente, lo usual es utilizar el conocimiento que se tiene de dónde podría haber una buena solución, pero, si no se tiene ninguna información puede partirse de cualquier valor y mejorarlo en el proceso de solución.
- Un procedimiento de búsqueda local parte de una solución inicial, calcula su vecindario N ( ) y escoge una nueva solución en ese vecindario, de acuerdo a un criterio de selección preestablecido. Es decir, realiza un movimiento que aplicado a da como resultado , este proceso se realiza reiteradamente.
- Se pueden definir diferentes criterios para seleccionar una nueva solución del entorno. Uno de los criterios más simples consiste en tomar la solución con mejor evaluación de la función objetivo, siempre que la nueva solución sea mejor que la actual. Este criterio, conocido como greedy (voraz, goloso), permite ir mejorando la solución actual mientras se pueda.

El algoritmo se detiene cuando la solución no puede ser mejorada o cuando se cumple el criterio de parada. El riesgo que se corre es el de haber encontrado una solución óptima local, no la solución global del problema.

Al aplicar un método de búsqueda cualquiera se puede presentar dos problemas:

1. El algoritmo puede ciclar, revisitando soluciones ya vistas, por lo que habría que introducir un mecanismo que lo impida.

2. El algoritmo podría iterar indefinidamente, por lo que se establece un criterio de parada.

Esta limitación de los métodos de búsqueda es el punto de inicio de muchas de las técnicas heurísticas, entre ellas la Búsqueda Tabú [24].

3.5.2 Conceptos de la búsqueda tabú

La búsqueda tabú puede ser aplicada directamente a expresiones verbales o simbólicas, sin necesidad de transformarlas a expresiones matemáticas, el requerimiento $x \in S$ (x es una solución factible) puede, por ejemplo, especificar condiciones lógicas o interconexiones que pueden ser difíciles de expresar matemáticamente por lo tanto será mejor dejarlas como expresiones verbales y codificarlas como reglas. Una solución pertenece al conjunto de élite dependiendo de su puntaje, el cual está relacionado con la función objetivo de la mejor solución encontrada durante la búsqueda, un óptimo local pertenece al conjunto de élite**.**

La característica que distingue a la Búsqueda Tabú de las otras heurísticas de búsqueda es el uso de la memoria la cual tiene una estructura basada en una lista tabú y unos mecanismos de selección del siguiente movimiento.

La lista tabú es una lista en el contexto computacional, donde se registran aquellas soluciones o atributos de soluciones que no deben ser elegidas.

Una forma sencilla de construir una lista tabú consiste en que cada vez que se realiza un movimiento, se introduce su inverso (si se pasó de a , el inverso es a) en una lista circular, de forma que los elementos de dicha lista están penalizados durante un cierto tiempo.

Por tanto, si un movimiento está en la lista tabú no será aceptado, aunque aparentemente sea mejor solución que la solución actual.

La lista tabú puede contener:

- Soluciones visitadas recientemente
- Movimientos realizados recientemente
- Atributos o características que tenían las soluciones visitadas.

Así como las costumbres sociales pueden cambiar con el tiempo, las soluciones tabú pueden dejar de serlo sobre la base de una memoria cambiante, debe haber una forma de “olvido estratégico”, es decir que una solución o atributo pueda salir de la lista tabú antes de que se cumpla su plazo. Esto se implementa a través del Criterio de aspiración, el cual permite que un movimiento sea admisible aunque esté clasificado como tabú.

Las aspiraciones son de dos clases:

- aspiraciones de movimiento y
- aspiraciones de atributo.

Una aspiración de movimiento, cuando se satisface, revoca la condición tabú del movimiento. Una aspiración de atributo, cuando se satisface revoca el status tabú del atributo. En éste último caso el movimiento puede o no cambiar su condición de tabú, dependiendo de sí la restricción tabú puede activarse por más de un atributo.

Se pueden tener los siguientes criterios de aspiración:

- **Aspiración por Default:** Si todos los movimientos posibles son clasificados como tabú, entonces se selecciona el movimiento “menos tabú”, es decir, si el movimiento está penalizado en la lista tabú durante 2 iteraciones más y está penalizado durante 1, es menos tabú que .
- **Aspiración por Objetivo:** Una aspiración de movimiento se satisface, permitiendo que un movimiento x sea un candidato para seleccionarse si, por ejemplo, F(x) < mejor costo, donde F(x) es la función objetivo. (en un problema de minimización).
- **Aspiración por Dirección de Búsqueda:** Un atributo de aspiración para la solución “s” se satisface si la dirección en “s” proporciona un mejoramiento

y el actual movimiento es un movimiento de mejora. Entonces “s” se considera un candidato.

Se define el concepto de movimiento por ser esencial para comprender el método de Búsqueda Tabú.

Se define un movimiento s como una función definida en un subconjunto X(s) de X s: X(s) → X. Por ejemplo, si el algoritmo inicializa en el nodo i, los movimientos posibles son aquellos arcos que unen el nodo con alguno de sus nodos adyacentes.

A $x \in X$ está asociado el conjunto N(x) (vecindario o entorno) el cual contiene los movimientos $s \in S$ que pueden ser aplicados a x, es decir $N(x) = \{s \in S: s \in X(x)\}$.

En la -ésima iteración, para evolucionar hacia otras soluciones, se seleccionan éstas en un entorno N( ), pero que no estén en la lista Tabú: (N( ) – {Lista Tabú}), evaluando cada una de las soluciones y quedándose con la mejor (que obviamente no es tabú).

Para realizar una búsqueda completa, es deseable que el tamaño del entorno no sea elevado. Si se considera un entorno de tamaño grande, con objeto de reducir el tiempo de computación, se puede realizar la búsqueda en un subconjunto tomado aleatoriamente [24].

Supongamos que el vecindario que se debe evaluar para determinar el siguiente movimiento es “grande” y es muy costoso en términos de tiempo de computación, evaluar todos y cada uno de los vecinos, se pueden usar algunos procedimientos aleatorios tales como los métodos de Monte Carlo que muestrean el espacio de búsqueda hasta terminar finalmente mediante alguna forma de limitación de iteración. En todo caso se debe determinar el número de muestras y el tamaño de las mismas de tal modo que no ocasione más tiempo de procesamiento que una revisión exhaustiva del vecindario [24].

3.5.3 Uso de la memoria

La búsqueda tabú se caracteriza porque utiliza una estrategia basada en el uso de estructuras de memoria para escapar de los óptimos locales, en los que se puede caer al “moverse” de una solución a otra por el espacio de soluciones.

Las estructuras de memoria usadas son de dos tipos:

Explícita: Cuando la solución se almacena de manera completa, se registran soluciones de élite visitadas durante la búsqueda. Por ejemplo { , } donde las son soluciones ocurridas en iteraciones anteriores. Una extensión de esta memoria explícita registra vecindarios áltamente atractivos pero inexplorados de las soluciones de élite.

De atributos: Se guarda información acerca de ciertos atributos de las soluciones pasadas, para propósitos de orientación de la búsqueda. Este tipo de memoria registra información acerca de los atributos o características que cambian al moverse de una solución a otra. Por ejemplo, en un grafo los atributos pueden consistir en nodos o arcos que son aumentados, eliminados o reposicionados por el mecanismo de movimientos.

La memoria, por lo tanto puede ser explícita, de atributos o ambas. En síntesis, la estructura de memoria explícita almacena soluciones de élite (que dan un óptimo local) y la memoria de atributos tiene como propósito guiar la búsqueda.

Es importante considerar que los métodos basados en búsqueda local requieren de la exploración de un gran número de soluciones en poco tiempo, por ello es crítico el reducir al mínimo el esfuerzo computacional de las operaciones que se realizan, lo que se puede conseguir registrando los atributos de las soluciones en vez de éstas para orientar la búsqueda más rápidamente.

La estructura de la memoria en la heurística de búsqueda tabú opera en relación a cuatro dimensiones principales:

- calidad
- influencia

- corto plazo (lo reciente)
- largo plazo (lo frecuente)

La calidad se refiere a la habilidad para diferenciar el mérito de las soluciones, identifica qué las hace tan buenas e incentiva la búsqueda para reforzar las acciones que conducen a una buena solución y desalienta aquellas que conducen a soluciones pobres.

La flexibilidad de la estructura de memoria permite que la búsqueda sea guiada en un contexto multiobjetivo, donde la bondad de una dirección de búsqueda particular puede estar determinada por más de una función, el concepto de calidad en la búsqueda tabú es más amplio que el usado implícitamente en los métodos de optimización, en los cuales se considera que un movimiento es de mejor calidad que otro porque produce una mejor “mejora” (tal es el caso del descenso más rápido), bajo el enfoque de búsqueda tabú un movimiento puede ser de mejor calidad si, por ejemplo, su frecuencia de ocurrencia en el pasado es baja o no ha ocurrido antes y nos permite explorar nuevas regiones. La definición de calidad de una solución es flexible y puede ser adaptado a la naturaleza del problema.

La dimensión, considera el impacto de las elecciones hechas durante la búsqueda, mide el grado de cambio inducido en la estructura de la solución o factibilidad, no sólo en calidad sino también en estructura, en un sentido la calidad puede entenderse como una forma de influencia. Registrar información acerca de las elecciones de un elemento de una solución particular incorpora un nivel adicional de aprendizaje, registra qué elementos o atributos generan ese impacto.

Esta noción puede ser ilustrada para el problema de distribuir objetos desigualmente pesados entre cajas, donde el objetivo es dar a cada caja, tan aproximadamente como sea posible, el mismo peso. Un movimiento que transfiere un objeto muy pesado de una es un movimiento de alta influencia, cambia significativamente la estructura de la solución actual, otro que intercambia objetos de pesos similares entre dos cajas no introduce mayor influencia. Tal movimiento puede no mejorar la solución actual, la solución actual es relativamente buena.

Se privilegian los movimientos etiquetados como influyentes, pero esto no quiere decir que no se opte en alguna iteración por uno poco influyente, pues éstos pueden ser tolerados si proporcionan mejores valores hasta que las mejoras a partir de ellos parezcan ser insignificantes. En tal punto, y en ausencia de movimientos de mejora, los criterios de aspiración cambian para dar a los movimientos influyentes un rango mayor y éstos puedan salir de la lista tabú antes del plazo establecido en su tabú.

En el problema de distribuir objetos desigualmente pesados entre cajas, donde el objetivo es dar a cada caja, tan aproximadamente como sea posible, el mismo peso, una solución de calidad puede ser aquella que hace que la diferencia de pesos de las cajas sea pequeña, es decir las cajas están aproximadamente balanceadas.

Estas dos dimensiones de la memoria: calidad e influencia, pueden estar consideradas de manera explícita en el algoritmo tabú, es decir, incluir una estructura de datos que registre la etiqueta de un movimiento o solución como de calidad y/o influyente paralela al registro de su condición tabú o no tabú; o puede estar dada de manera implícita, como considerar que es de calidad si es un óptimo local y/o considerar que es influyente si permite diversificar la búsqueda. Sin embargo las otras dos dimensiones: memoria de corto plazo y la de largo plazo, siempre van expresadas de manera explícita por medio de la lista tabú y de la tabla de frecuencias.

Una distinción importante en la búsqueda tabú es la de diferenciar la memoria de corto plazo y la de largo plazo, cada tipo de memoria tiene sus propias estrategias, sin embargo, el efecto de la utilización de ambos tipos de memoria es el mismo: modificar el vecindario N(x) de la solución actual x (convertirlo en un nuevo vecindario N’(x)).

En la memoria de corto plazo, el vecindario es generalmente:

N’(x) = {N(x) – Lista tabú}

En las estrategias que usan la memoria de largo plazo N'(x) puede ser expandido para incluir otras soluciones que no están en N(x).

En ambos casos el vecindario de x, N(x) no es un conjunto estático sino un conjunto que varía dinámicamente de acuerdo a la historia de la solución x. En esencia la memoria de corto plazo utiliza la estructura tabú para penalizar la búsqueda y la memoria de largo plazo utiliza las frecuencias para determinar si un movimiento no tabú puede ser elegido o no [24].

3.5.4 Memoria basada en lo reciente (corto plazo)

Es una "memoria" donde se almacenan los últimos movimientos realizados, y que puede ser utilizada para "recordar" aquellos movimientos que hacen caer de nuevo en soluciones ya exploradas. Su objetivo es penalizar la búsqueda para evitar el ciclado. Es una manera de definir el entorno o vecindario reducido de una solución, consiste en etiquetar como tabú las soluciones previamente visitadas en un pasado cercano, ciertos movimientos se consideran tabú, de forma que no serán aceptados durante un cierto tiempo o un cierto número de iteraciones, se considera que tras un cierto número de iteraciones la búsqueda está en una región distinta y puede liberarse del status tabú [24].

3.5.5 Memoria Basada en Frecuencia (largo plazo)

La memoria basada en frecuencia proporciona un tipo de información que complementa la información proporcionada por la memoria basada en lo reciente, ampliando la base para seleccionar movimientos preferidos. Como lo reciente, la frecuencia a menudo toma en cuenta las dimensiones de calidad de la solución e influencia del movimiento. En esta estructura de memoria se registra la frecuencia de ocurrencias de los movimientos, las soluciones o sus atributos y puede ser:

- **Frecuencia de transiciones**: Cantidad de veces que una solución es la mejor o cantidad de veces que un atributo pertenece a una solución generada.
- **Frecuencia de residencia**: Cantidad de iteraciones durante la cual un atributo pertenece a la solución generada.

Una alta frecuencia de residencia, por ejemplo, puede indicar que un atributo es altamente atractivo si S es una secuencia de soluciones de alta calidad, o puede indicar lo contrario si S es una secuencia de soluciones de baja calidad. Por otro lado, una frecuencia de residencia que es alta (baja) cuando S contiene tanto soluciones de alta como de baja calidad puede apuntar a un atributo fortalecido (o excluido) que restringe al espacio de búsqueda, y que necesita ser desechado (o incorporado) para permitir diversidad.

La memoria a largo plazo tiene dos estrategias asociadas:

- Intensificar y
- Diversificar la búsqueda.

La intensificación consiste en regresar a regiones ya exploradas para estudiarlas más a fondo. Para ello se favorece la aparición de aquellos atributos asociados a buenas soluciones encontradas. Para evitar regresar a óptimos locales cada cierto número de iteraciones, la búsqueda tabú utiliza además otra estrategia, como es la diversificación.

La Diversificación consiste en visitar nuevas áreas no exploradas del espacio de soluciones. Para ello se modifican las reglas de elección para incorporar a las soluciones atributos que no han sido usados frecuentemente. Una forma clásica de diversificación consiste en reiniciar periódicamente la búsqueda desde puntos elegidos aleatoriamente, si se tiene alguna información acerca de la región factible se puede hacer un “muestreo” para cubrir la región en lo posible, si no, cada vez se escoge aleatoriamente un punto de partida (método multi arranque).

Otro método para diversificar propone registrar los atributos de los movimientos más utilizados en los anteriores movimientos, penalizándolos a través de una lista tabú a largo plazo y así explorar otros entornos.

El uso racional de estas dos estrategias da lugar a dos características de la búsqueda tabú, el reencadenamiento de trayectorias y la oscilación estratégica.

Una integración de las estrategias de intensificación y diversificación consiste en el reencadenamiento de trayectorias.

Este acercamiento genera soluciones nuevas explorando trayectorias que "conectan" soluciones de élite (óptimos locales) comenzando desde una de esas soluciones; llamada solución inicial, y generando un camino en el vecindario que conduce a otras soluciones, llamadas soluciones guía. En una colección dada de soluciones de élite, el papel de la solución inicial y una solución guía se pueden alternar. Esto es, se puede generar simultáneamente un conjunto de soluciones actuales, extendiendo caminos diferentes, y permitiendo que una solución inicial sea reemplazada (como una solución orientadora para las otras) por otra, si ésta satisface un criterio de aspiración suficientemente fuerte. Debido a que sus papeles son intercambiables, la solución inicial y la guía son llamadas soluciones de referencia.

El proceso de generar caminos entre soluciones de referencia está acompañado por movimientos de selección que introducen atributos contenidos en las soluciones que operan como soluciones guías.

El proceso también puede continuar más allá de una solución de referencia cambiando el criterio de selección de movimiento que estratégicamente introducen atributos que no están en las soluciones guías.

En ambos casos, estos atributos están ponderados para determinar a cuáles movimientos se les da una prioridad superior. La generación de tales caminos dentro de un vecindario "reasocia" puntos previos en formas diferentes a las registradas en la historia previa de búsqueda.

La oscilación estratégica guía los movimientos hasta que se llega a un límite que por lo general representaría un punto donde el método debe parar. En lugar de parar, se le permite al procedimiento cruzar ese límite modificando la definición de entorno y el criterio de evaluación, La repetición de este proceso oscilatorio de cruce de límite proporciona un marco adecuado para combinar estrategias de intensificación y diversificación.

Una ventaja de esta estrategia es que al moverse fuera de la frontera y retornar desde una diferente dirección se descubren oportunidades para mejoramiento que no son alcanzables cuando la búsqueda está confinada de una manera estrecha [10].

3.5.6 Metodología de la búsqueda tabú

La búsqueda tabú procede como cualquier algoritmo de búsqueda: Dada una solución x se define un entorno o vecindario N(x), se evalúa y se "mueve" a una mejor solución pero, en lugar de considerar todo el entorno o vecindario la búsqueda tabú define el entorno reducido N*(x) como aquellas soluciones disponibles (no tabú) del entorno de x.

3.5.7 Algoritmo de la búsqueda tabú simple

```
Generar solución inicial

 :=  .

 =   . (  es la solución actual)

MIENTRAS la condición de finalización no se encuentre

HACER:

Identificar N( ). (Vecindario de  )

Identificar T(   ). (Lista Tabú)

Identificar A(   ). (Conjunto de Aspirantes)

Determinar N*(   ) = {N( ) – T(   )} ∪A(   ). (Vecindario reducido)

Escoger la mejor x ∈ N*(   )

"Guardar" x si mejora la mejor solución conocida   :=  .

Actualizar la lista tabú

 :=     .

FIN MIENTRAS
```

Al confeccionar la Lista Tabú se debe considerar:

- Si sus elementos son soluciones completas o atributos.
- Tamaño de la lista tabú (tabú tenure).
- Si se usara memoria de atributos, elección de los atributos para almacenar en la lista tabú.
- Establecer un criterio mediante el cual un movimiento tabú pueda ser aceptado (nivel de aspiración).

¿Qué se necesita para su implementación?

Algo que es básico e indispensable es que el solucionador del problema, el que usará la heurística de búsqueda tabú, esté familiarizado con el problema, conozca su naturaleza, la forma de las soluciones factibles de tal manera que pueda sugerir la configuración de la lista tabú, del vecindario y el criterio de aspiración [24].

CAPÍTULO 4 COMPARACIÓN DE ALGUNOS ALGORITMOS BASADOS EN HEURÍSTICAS

En este capítulo vamos a presentar la solución al problema de la mochila con diferentes algoritmos explicados ya en los capítulos anteriores. Se simularan los vectores de peso y costo aleatorios y se correrán los algoritmos en MATLAB presentes en el apéndice. Cada uno de ellos nos proporcionará el número de iteraciones y el tiempo que tardarón para llegar a la solución. Con esta información se construirá una tabla en la que podremos comparar el desempeño de dichos algoritmos. Finalmente se hará un análisis de los resultados para presentar una conclusión sobre estos algoritmos [3].

Ejemplo 4.1: Vamos a empezar por la solución de un problema clásico de la mochila, el cual fue tomado de [3]. La mochila es capaz de soportar un peso de 60 kg, utilizando solo 5 objetos cuyos vectores de costos (P) y pesos (W) son los siguientes:

P= [122 2 144 133 52];

W= [63 1 71 73 24];

Tabla 4.1

algoritmo	tiempo	número de iteraciones	Costo óptimo	Peso alcanzado
backtrack1	0.0704	32	$54	25 kg
backtrack2	0.0312	4	$ 54	25 kg
backtrack3	0.1618	4	$ 54	25 kg
saks	0.0153	2	$52	24 kg
hillks	0.0084	10	$54	25 kg
tabuks	0.0089	20	$54	25 kg

Se puede observar en la tabla 4.1, que todos los algoritmos excepto saks, tuvieron un costo óptimo de $54 y un peso alcanzado de 25 kg, de los cuales, hillks fue quien menos tiempo tardó para llegar a la solución óptima. Por otro lado el algoritmo que más tiempo tardó en llegar a la solución óptima fue bactrack3. Para este ejemplo en particular, saks fue el algoritmo que menos iteraciones ejecutó y

bactrack1 fue el que más iteraciones realizó. Por lo tanto, se puede concluir que hillks tuvo el mejor desempeño pues alcanzó el óptimo en el menor tiempo posible.

Este problema podría resolverse a mano, simplemente eligiendo y sumando los pesos y costos de aquellos objetos que no sobrepasen el peso que es capaz de soportar la mochila.

4.1 Aplicaciones ilustrativas del problema de la mochila

En esta sección se presentan aplicaciones del problema de la mochila. Algunas de estas aplicaciones se refieren a la formulación directa del problema, dentro del formato aceptable de modelos de programación matemática, en particular de la programación entera.

4.1.1 Problema de CONASUPO (DICONSA)

Una camioneta de 900 kilos de capacidad de la distribuidora CONASUPO, S.A (DICONSA), visita un determinado número de poblados rurales, cada uno en su día de tianguis. La camioneta puede llevar café, maíz, frijol y arroz. El café viene empaquetado en un saco de 200 kg, el maíz en uno de 400 kg, el frijol en uno de 500 kg y el arroz en uno de 300 kg. Como esos productos se venden subsidiados a las poblaciones rurales, el ahorro para el consumidor rural, ya estimado por producto, es el siguiente [22]:

Tabla 4.2

producto	ahorro	peso
café	$ 50	200 kg
maíz	$120	400 kg
frijol	$170	500 kg
Arroz	$ 80	300 kg

Como los paquetes de los productos agrícolas son indivisibles (por ejemplo no se puede tener un bulto de 93 kilos de arroz, o 327 kilos de maíz), y la capacidad de la camioneta está limitada a 900 kilos, se pregunta, ¿cuál debe ser el cargamento que maximiza el ahorro estimado para el consumidor rural?

Matemáticamente se tiene que si:

Entonces el problema es:

Si , en vez de ser entero, es una variable continua, se pueden cargar los productos agrícolas en cualquier cantidad, este problema se resuelve por el *método simplex.* Como la variable es entera, el problema se resuelve por los métodos descritos en los capítulos anteriores [22].

Observando la tabla 4.2, podemos decir que los vectores de pesos y costos, que soporta la camioneta de 900 kg son:
P = [50 120 170 80];
W= [200 400 500 300];
Los resultados arrojados por los algoritmos para este problema se presentan a continuación:

Tabla 4.3

algoritmo	**tiempo**	**Número de iteraciones**	**Costo óptimo**	**Peso alcanzado**
backtrack1	0.0665	16	$ 290	900 kg
backtrack2	0.1126	12	$ 290	900 kg
backtrack3	0.0331	1	$ 290	900 kg
saks	0.3798	18	$ 250	900 kg

hillks	0.0299	60	$ 290	900 kg
tabuks	0.0691	34	$ 290	900 kg

Analizando la tabla 4.3, nos podemos percatar de que hillks vuelve a ser el algoritmo que menos tiempo realizó, pero en esta ocasión también es el que más iteraciones utilizó. Por otro lado, podemos ver que bactrack3 es el algoritmo que menos iteraciones realizó en el segundo mejor tiempo y que alcanzó el óptimo, por lo que concluimos que es el mejor que se desempeñó en este ejemplo. Saks vuelve a ser el único que se alejó del costo óptimo y más aún, el que más tiempo se tardó en aproximarse a la solución óptima.

Existen varias variantes a este problema, por ejemplo:

a) Deben existir cotas inferiores en las variables. Esto quiere decir, por ejemplo, que la camioneta debe llevar por lo menos un bulto de cada producto agrícola.
b) Las variables de decisión pueden tomar únicamente los valores enteros cero o uno. Toman el valor cero cuando no se incluye el bulto del producto y uno, cuando sí se incluye. Sin embargo, no se puede cargar más de un bulto por artículo.
c) Todas estas variantes pueden resolverse por medio de *la programación entera* u otros métodos. Estos tipos de problemas enteros, reciben el nombre de problemas **"mochila"**.

4.1.2 Problema del presupuesto de capital.

Un grupo financiero tiene 5 proyectos de inversión. Cada proyecto , necesita de una inversión de millones de pesos, y se pronostica que ese proyecto rendirá millones de pesos anuales de utilidad cuando el proyecto esté funcionando. La capacidad total de inversión es de 91 millones de pesos. La tabla 4.4 resume los datos asociados a cada proyecto de inversión [22]:

Tabla 4.4

Proyecto número	*Inversión en millones de pesos*	*Retorno anual de la Inversión en millones de pesos*
1	36	54
2	24	18
3	30	60
4	32	32
5	26	13

Se trata de decidir cuál de los cinco proyectos debe ejecutarse en el periodo de planeación de tres años. En este sentido, el problema se reduce a una decisión del tipo “si-no” en cada proyecto. Esta decisión está codificada en forma numérica con una variable binaria, donde el valor 1 representa “si el proyecto se acepta” y el valor 0 representa “el proyecto es rechazado”. El problema de decisión lo formalizamos definiendo la variable para que represente el proyecto El modelo asociado resulta entonces:

Donde la capacidad de la “mochila” ahora es $91 millones de pesos y la inversión en millones representa el vector de “pesos”, el retorno anual de inversión en millones representa ahora el vector de “costos”, los cuales vienen dados por:

P= [54 18 60 32 13];

W= [36 24 30 32 26];

Y cuya solución, para este problema viene dada en la tabla 4.5

Tabla 4.5

algoritmo	tiempo	Número de iteraciones	Costo óptimo en millones	Retorno óptimo en millones
backtrack1	0.0086	32	$ 90	$ 132
backtrack2	0.0263	22	$ 90	$ 132
backtrack3	0.1613	1	$ 90	$ 132
saks	0.4274	12	$ 82	$ 63
hillks	0.1026	29	$ 90	$ 132
tabuks	0.0795	31	$ 90	$ 132

La solución entera óptima para el modelo es , , , , con y millones. Es decir los proyectos 1, 2, 3 son los que el grupo financiero en su toma de decisiones elegirán para la empresa, como esta solución es factible también es óptima para este problema tipo mochila cero-uno [22].

Como podemos ver en la tabla 4.5, y en las tablas anteriores, todos los algoritmos, excepto saks, alcanzaron la misma solución óptima y el mismo peso, otra observación importante que podemos mencionar es que hillks ya no tuvo el mejor tiempo de solución, sino que ahora backtrack1 fue el que realizó menos tiempo, sin embargo, también fue el algoritmo que más iteraciones realizó y el algoritmo que menos iteraciones ejecutó fue backtrack3. En este ejemplo, es difícil decidir qué algoritmo tuvo el mejor desempeño pues los que tardaron menos tiempo hicieron más iteraciones.

En general hemos visto que los resultados de los algoritmos heurísticos, reportados en las tablas anteriores para problemas de la mochila relativamente pequeños, nos han arrojado “diferentes” resultados en cuanto a soluciones óptimas, peso alcanzado, tiempo de solución o búsqueda y número de iteraciones.

Podríamos concluir que hillks, en la mayoría de los casos, fue el algoritmo que menos tiempo realizó para encontrar la solución, backtrack3 el que menos iteraciones realizó. Pero algo muy importante y que no podíamos dejar de mencionar es que los algoritmos para problemas de la mochila relativamente pequeños alcanzaron la solución en poco tiempo y con un número pequeño de iteraciones, comparadas con las que veremos más adelante.

Ahora estudiaremos el comportamiento de estos algoritmos heurísticos para algunos ejemplos de problemas de la mochila relativamente grandes. ¿Se comportarán igual en cuanto a tiempo de solución, iteraciones y soluciones óptimas? Y lo más importante ¿se ciclarán estos algoritmos a la hora de correrlos? Esas preguntas se responderán en la siguiente sección.

4.2 Aplicación del problema de la mochila con ejemplos numéricos

Generamos dos vectores aleatorios, cuyas componentes varían en el intervalo (1,500) y las cuales representan los costos y los pesos de un problema de la mochila, de la siguiente forma:

```
a=1;

b=500;

P=a+ (b-a)*rand (1, 20)
```

Lo anterior quiere decir que se generará un vector de costos (P), aleatorio, con componentes en el intervalo abierto (1,500), donde P es el vector renglón que representa el valor de 20 objetos, análogamente se procede para generar los vectores de pesos (W). Una vez que MATLAB haya generado los vectores aleatorios se pegan y se corren en cada uno de los algoritmos.

Ejemplo 4.2.1: Vamos a empezar por un problema tipo mochila de 20 objetos, que es capaz de soportar un peso de 4000 kg, cuyos vectores de costos (P) y pesos (W) son los siguientes:

```
P=[407.5471   452.9902    64.3664   456.7746   316.5473    49.6727
   139.9706   273.8939   478.7959   482.4794    79.6489   485.3258
   478.6263   243.2024   400.3400    71.8013   211.4589   457.9520
   396.3115   479.7867];

W=[328.2146    18.8201   424.7155   467.0626   339.6888   379.1123
   371.8231   196.7213   328.0835    86.4222   353.3170    16.8846
   139.1846    24.0395    49.4688   411.9055   347.7195   159.2326
   475.1608    18.1886];
```

La solución de este problema con los algoritmos anteriores se resume a continuación:

Tabla 4.6

algoritmo	Tiempo segundos	Número de iteraciones	Costo óptimo	peso alcanzado
backtrack1	42.6519	1048576	6,241.7	3,720.2
backtrack2	62.0929	1039119	6,241.7	3,720.2
backtrack3	0.1720	3	6,241.7	3,720.2
saks	0.2716	68	5,034.4	4,273.7
hillks	0.0753	29	6,241.7	3,720.2
tabuks	0.0123	58	6,241.7	3,720.2

En este problema de la mochila de 20 objetos, cuyos costos y pesos son mayores que en ejemplos anteriores, podemos darnos cuenta que nuevamente todos los algoritmos excepto saks, alcanzaron la misma solución óptima y el mismo peso. Otra observación importante que podemos mencionar es que hillks ya no tuvo el mejor tiempo de solución, sino que ahora tabuks dio la solución óptima en el menor tiempo posible, sin embargo no dio el menor número de iteraciones sino que fue backtrack3 quien realizó el menor número de iteraciones. Es importante mencionar que los algoritmos backtrack1 y bactrack2 dieron la solución óptima en

un tiempo e iteraciones muy grandes, en comparación con los demás algoritmos. En este ejemplo, consideramos que backtrack3 tuvo el mejor desempeño.

Pasemos ahora a ver otro ejemplo del problema de la mochila más grande cuyos vectores de peso y costo, también fueron simulados o creados aleatoriamente en la ventana de comandos de MATLAB.

Ejemplo 4.2.2: se quiere llenar una mochila con 40 objetos, que es capaz de soportar un peso de 2500 kg. Y los vectores de costo y peso de los objetos son los siguientes:

```
P=[407.5471   452.9902    64.3664   456.7746   316.5473    49.6727
   139.9706   273.8939   478.7959   482.4794    79.6489   485.3258
   478.6263   243.2024   400.3400    71.8013   211.4589   457.9520
   396.3115   479.7867   328.2146    18.8201   424.7155   467.0626
   339.6888   379.1123   371.8231   196.7213   328.0835    86.4222
   353.3170    16.8846   139.1846    24.0395    49.4688   411.9055
   347.7195   159.2326   475.1608    18.1886];
```

```
W=[219.9334   191.3977   382.9929   397.8048    94.2494   245.3924
   223.3475   323.5102   354.9731   377.5887   138.7365   340.1716
   327.8939    82.1433    60.3798   249.6837   479.9122   170.8525
   293.0486   112.6822   375.8823   128.2925   253.4726   349.8393
   445.5607   479.6864   274.0605    70.1736    75.4977   129.4966
   420.5179   127.8868   407.3281   122.5190   464.7025   175.6419
    99.1010   126.2908   308.4063   237.1711];
```

Las soluciones arrojadas por los algoritmos se presentan en la siguiente tabla:

Tabla 4.7

algoritmo	Tiempo segundos	Número de iteraciones	Costo óptimo	peso alcanzado
backtrack1	No se reportó	No se reportó	No se reportó	No se reportó
backtrack2	No se reportó	No se reportó	No se reportó	No se reportó
backtrack3	0.2980	2	5,710.8	2,481.1
saks	0.1201	75	3,627.8	2,433.6
hillks	0.4120	39	4,533.6	2,478.2
tabuks	0.1168	58	4,533.6	2,478.2

Al observar en la tabla 4.7, podemos decir que solamente backtrack 3 alcanzó el peso máximo y la solución óptima, siendo también el que realizó menos iteraciones. Por otra parte, tabuks es el que menor tiempo ocupó pero sin llegar a la solución óptima y saks obtuvo el menor costo y peso, en comparación con los demás algoritmos. Es importante mencionar que los algoritmos backtrack1 y backtrack2 no reportaron costo óptimo ni peso alcanzado porque se detuvieron después de correrlos una hora aproximadamente. En este ejemplo es claro que backtrack3 es el que tuvo mejor desempeño.

Finalicemos esta sección, con este último ejemplo, ahora con un problema de la mochila “más grande” que los ejemplos anteriores, no perdiendo de vista que sus vectores de pesos y costos también se generaron aleatoriamente.

Ejemplo 4.2.3: se va a llenar una mochila con 60 objetos, que es capaz de soportar un peso de 5000 kg y cuyos vectores de costo y peso de los objetos son los siguientes:

P=[407.5471 452.9902 64.3664 456.7746 316.5473 49.6727
139.9706 273.8939 478.7959 482.4794 79.6489 485.3258
478.6263 243.2024 400.3400 71.8013 211.4589 457.9520
396.3115 479.7867 328.2146 18.8201 424.7155 467.0626
339.6888 379.1123 371.8231 196.7213 328.0835 86.4222
353.3170 16.8846 139.1846 24.0395 49.4688 411.9055
347.7195 159.2326 475.1608 18.1886 219.9334 191.3977

382.9929 397.8048 94.2494 245.3924 223.3475 323.5102
354.9731 377.5887 138.7365 340.1716 327.8939 82.1433
60.3798 249.6837 479.9122 170.8525 293.0486 112.6822];

W=[375.8823 128.2925 253.4726 349.8393 445.5607 479.6864
274.0605 70.1736 75.4977 129.4966 420.5179 127.8868
407.3281 122.5190 464.7025 175.6419 99.1010 126.2908
308.4063 237.1711 176.4781 415.5835 293.0468 275.3121
458.6796 143.6337 378.8429 377.1108 190.8425 284.3430
38.8513 27.9211 265.8680 389.8044 467.0713 65.8232
284.8430 235.2259 6.9391 169.2242 81.9290 397.3480
156.2963 264.7380 83.6587 301.3890 132.2227 327.3855
344.9180 374.3276 225.8203 42.8269 115.2595 456.7553
77.0366 413.0827 269.6329 498.0712 40.0096 221.8965];

La información de las soluciones, dadas por los algoritmos se describen en la siguiente tabla:

Tabla 4.8

algoritmo	**Tiempo segundos**	**Número de iteraciones**	**Costo óptimo**	**peso alcanzado**
backtrack1	No se reportó	No se reportó	No se reportó	No se reportó
backtrack2	No se reportó	No se reportó	No se reportó	No se reportó
backtrack3	0.2560	2	11,250.2	4,949.3
saks	0.0959	65	4,659.8	4,981.2
hillks	0.0108	63	9,498.3	4,998.2
tabuks	0.0062	58	9,498.3	4,998.2

De la tabla 4.8, otra vez podemos decir que solamente backtrack 3 dio el costo óptimo, con un peso menor al solicitado, además nuevamente backtrack3 fue el algoritmo que menos iteraciones realizó. Por otra parte, tabuks y hillks vuelven a tener el menor tiempo pero sin llegar a la solución. Es importante mencionar que saks tuvo un costo óptimo y peso alcanzado, más "alejado" o "bajo" en comparación con los demás algoritmos y que los algoritmos backtrack1 y backtrack2, no reportaron costo óptimo ni peso alcanzado, lo cual nos hace sospechar que backtrack1 y bactrack2 se ciclan con problemas de la mochila relativamente grandes.

4.3 Calibración de parámetros en el algoritmo saks

La calibración de parámetros es una técnica que consiste en probar diferentes valores para los parámetros en los problemas de prueba y seleccionar aquellos que arrojen mejores resultados. Las desventajas de usar la calibración de parámetros se puede resumir en los siguientes puntos [11]:

- La calibración de parámetros consume mucho tiempo de cómputo, a pesar de que los parámetros sean optimizados uno por uno dejando un lado las iteraciones.
- Los valores de los parámetros seleccionados no son necesariamente óptimos a pesar del esfuerzo computacional invertido.
- Intentar probar todas las posibles combinaciones es prácticamente imposible.

Esta técnica computacional se usará para calibrar los parámetros, alpha (parámetro de refrigeración) y T_0 (temperatura inicial), correspondientes al algoritmo saks con el objetivo de dar una mejor solución óptima al problema de la mochila, pues el algoritmo saks, en comparación con los demás, proporcionó la solución óptima menor. Para esta calibración tomaremos alpha y T_0

.

Analizando las tablas de la sección anterior podemos decir que el algoritmo saks, obtuvo la solución óptima más alejada y en algunos problemas no tuvo el mejor desempeño en cuanto tiempo, número de iteraciones y peso alcanzado, respecto a los demás algoritmos. En la tabla 4.9 se resume la solución del algoritmo saks al problema de la mochila para los diferentes problemas que se presentaron en la sección anterior cuyos parámetros fueron alpha=0.1, T_0=45, los cuales se tomaron arbitrariamente.

Tabla 4.9 Soluciones del algoritmo saks con alpha=0.1 y T_0=45.

Tablas	Tiempo	Número de iteraciones	Costo óptimo en pesos	Peso alcanzado en kilogramos	Peso máximo	Número de objetos
4.1	0.0153	2	52	24	25	5
4.3	0.3798	18	250	900	900	4
4.5	0.4274	12	63	82	91	5
4.6	0.2716	68	5,034.4	4,273.7	4000	20
4.7	0.1201	75	3,627.8	2,433.6	2500	40
4.8	0.0959	65	4,659.8	4,981.2	5000	60

Una vez probados diferentes valores para los parámetros, alpha y T_0 en el algoritmo saks, el cual se puede consultar en el apéndice, elegimos alpha=.999 y T_0=10000, porque fueron los valores con los que se obtuvo la mejor solución del problema de la mochila de cada uno de los ejemplos resueltos anteriormente, ver la tabla 4.10.

Tabla 4.10 resultados del algoritmo saks con alpha=0.999 y T_0=10000

tablas	tiempo	Número de iteraciones	Costo óptimo	Peso alcanzado	Peso máximo	Número de objetos
4.1	0.0553	35	54	25	25	5
4.3	0.0040	35	290	900	900	4
4.5	0.0034	45	132	90	91	5
4.6	0.0038	45	4,333.9	3,238.7	4000	20
4.7	0.0915	60	3,344.1	2,460.1	2500	40
4.8	0.0183	58	7,000	4,548.5	5000	60

En la tabla 4.10 podemos observar que el algoritmo saks obtuvo un peso y una solución óptima mejor inclusive igual que la alcanzada con los demás algoritmos, con unas iteraciones y un tiempo también mejores, aunque es importante recalcar que esto sucedió para problemas de la mochila relativamente pequeños.

Ahora, es importante observar que en dos de los tres ejemplos del problema de la mochila, relativamente grandes, el algoritmo saks volvió a quedar alejado de la solución óptima, aunque los resolvió en un tiempo y un número de iteraciones bastante buenos.

No hay reglas generales para elegir el mejor “programa de enfriamiento” o conjunto de parámetros, el resultado final depende del cambio en la función objetivo debido a los movimientos que se utilizan en el espacio de búsqueda.

Conclusiones

En esta tesis, se ha propuesto comparar el desempeño de algoritmos heurísticos, para resolver el problema de la mochila, basados en: vuelta atrás (backtrack1, backtrack2 y backtrack3), búsqueda tabú (tabuks), la escalada (hillks) y recocido simulado (saks).

Después de probar los algoritmos arriba mencionados con ejemplos particulares, tomados de la literatura y con ejemplos numéricos aleatorios, y observando las tablas de los resultados arrojados por los algoritmos, concluimos lo siguiente: Los algoritmos backtrack1 y backtrack2 basados en vuelta atrás (backtracking), los cuales implementan la fuerza bruta y árbol de poda respectivamente, se han comportado de manera exitosa en 4 de los 6 ejemplos, los cuales corresponden con problemas de la mochila relativamente pequeños, pero se comportaron de manera deficiente, en 2 de los 6 ejemplos, los cuales corresponden a problemas de la mochila relativamente grandes. Por otro lado, el algoritmo backtrack3, basado también en vuelta atrás (backtracking) implementando la poda sobre la base de la función de delimitación, se ha comportado de manera exitosa en los 6 ejemplos del problema de la mochila, pues este algoritmo dio un peso alcanzado y una solución mayores, en comparación con los demás algoritmos, además de que fue el algoritmo que utilizó el menor número de iteraciones para encontrar la solución óptima. De lo cual podemos concluir que, backtrack3 es el que mejor desempeño tuvo en todos los ejemplos.

El algoritmo saks, basado en recocido simulado (simulated annealing), en los 6 ejemplos, fue el algoritmo que dio una solución y un peso más bajos, en comparación con los demás algoritmos, aunque se ubica entre los 3 primeros lugares en cuanto el tiempo de solución, pero también, entre los 3 primeros lugares en cuanto al mayor número de iteraciones. Por lo tanto concluimos que saks es el algoritmo que tuvo el peor desempeño.

En la sección 4.3 presentamos una calibración de los parámetros del algoritmo saks que lo convierte en un algoritmo competente al igual que los demás en cuanto a peso alcanzado, solución óptima, tiempo de solución e iteraciones.

El algoritmo hillks, basado en colina inclinada (hill climbing), se comportó eficientemente en 2 de los 3 ejemplos pequeños del problema de la mochila al alcanzar el peso y la solución y con un buen tiempo de solución, pero no así en el número de iteraciones. Por otra parte, observamos que para problemas de la mochila grandes se aleja del valor óptimo, con un buen tiempo de solución pero no así en el número de iteraciones.

Por último, el algoritmo tabuks, basado en búsqueda tabú (tabu search), para problemas de la mochila relativamente pequeños, obtuvo soluciones óptimas y pesos aceptables, además con uno de los mejores tiempos, aunque fue de los algoritmos que más iteraciones realizó. Es importante mencionar que para problemas de la mochila grandes, se alejó un poco de la solución óptima y de los pesos, en comparación con backtrack3, aunque fue el algoritmo que menor tiempo realizó en este tipo de problemas. Con lo mencionado anteriormente se concluye que es el algoritmo que reportó el menor tiempo en problemas de la mochila grandes, aproximándose a un peso y una solución óptima bastante buena o aceptable. Para problemas de la mochila pequeños fue uno de los algoritmos que dio un peso y una solución mayores, aunque no en el menor tiempo. Por lo anterior podemos concluir que tabuks se comportó de manera exitosa en problemas de la mochila tanto pequeños como grandes.

Para todo aquel que esté interesado en la programación matemática u optimización heurística cabe la posibilidad de dotar a los algoritmos heurísticos presentados en esta tesis de cambios, como se hizo en la calibración de parámetros de saks, que permitan no solo un mejor tratamiento integral de un problema de optimización, sino también de nuevas propuestas para abordar varios problemas conocidos de la optimización, dando así lugar a futuras investigaciones sobre la optimización heurística, la cual es una herramienta muy poderosa para diferentes áreas de las ciencias, para resolver distintos problemas de la vida real

Apéndice

El objetivo de este apéndice es presentar, la implementación de los algoritmos heurísticos, programados en MATLAB que cubre los aspectos más sofisticados de diseño y análisis de algoritmos. De los cuales se presentan seis algoritmos de búsqueda heurística para la solución del problema de la mochila. Para una mejor comprensión de estos algoritmos le sugerimos consulte [3].

A-1 Algoritmo backtrack1 (fuerza bruta)

Esta implementación es, una simple aplicación de retroceso aplicado al problema de la mochila sin "trucos inteligentes". En esta implementación se explorarán todas las posibles soluciones al problema sin tratar de podar el espacio de estados.

```
function [optP,optX,r]=backtrack1(l,x,w,p,optP,optX,M,r)

% Invocación: [optP,optX,r]=backtrack1(l,x,w,p,optP,optX,M,r)
% Parámetros: l: el nivel recursivo, para inicializar 1
%             x: valor actual de x, inicializado desde cero
%             w: el vector de pesos
%             p: el vector de costos
%          optP: el beneficio óptimo, inicializa en 0
%          optX: el valor óptimo para X, inicializa en []
%             M: el peso máximo
%             r: número de llamadas a la función, iniciando en 1
% Esta función implementa fuerza bruta para resolver backtracking
% problema de la mochila.
% derechos del autor Lars Aurdal / Riskshopitalet

% obtiene la longitud del vector
  n=length (w);

  % si en la parte inferior de la recursión
  if (l>n)
    r=r+1;    % incrementa el número de llamadas

    % calcula el peso actual
     tmpW=0;
    for (i=1: n)
      tmpW=tmpW+w (i)*x (i);
    end

    % si el peso actual es inferior al peso máximo
```

```
    % calcular la ganancia
    if (tmpW <= M)
      tmpP=0;
      for (i=1: n)
    tmpP=tmpP+p (i)*x (i);
      end

      % si el beneficio actual supera al beneficio máximo actual
       if (tmpP>optP)
    optP=tmpP;
    optX=x;
      end
    end
  else

    % si no es así, en el fondo de la recursividad empezar a explorar
    % la 1-rama
     x (l) =1;
    [optP, optX, r]=backtrack1 (l+1, x, w, p, optP, optX, M, r);

    % luego explore la 0-rama
    x (l) =0;
    [OptP, optX, r]=backtrack1 (l+1, x, w, p, optP, optX, M, r);
  end
```

A-2 Algoritmo backtrack2 (poda del árbol)

Esta implementación es, backtracking aplicado al problema de la mochila con la poda simple. Un nodo se excluye de la consideración adicional si no puede ser la raíz de una solución admitida.

```
function [optP,optX,r]=backtrack2(l,CurW,x,w,p,optP,optX,M,r)

% Invocation: [optP,optX,r]=backtrack2(l,CurW,x,w,p,optP,optX,M,r)
% parámetros: l: el nivel recursivo, inicializa en 1
%          curW: el peso actual, inicializa en 0
%             x: el valor actual de x, inicializa en ceros
%             w: el vector de pesos
%             p: el vector de costos
%          optP: el beneficio óptimo, inicializa en 0
%          optX: el valor óptimo para X, inicializa en []
%             M: el máximo peso
%             r: número de llamadas a la función iniciada en 1
% Esta función implementa backtracking, de forma sencilla
% la poda para resolver el problema de la mochila.
% derechos de autor Lars Aurdal / Riskshopitalet

  % obtiene la longitud del vector
  n=length (w);
```

```
  % si en la parte inferior de la recursión
  if (l>n)
    r=r+1;    % incrementar el número de llamadas

    % calcule el beneficio actual
    tmpP=0;
    for (i=1: n)
      tmpP=tmpP+p (i)*x (i);
    end

    % si el beneficio actual supera al beneficio máximo
    % a continuación, actualice máximo beneficio y optX
     if (tmpP > optP)
      optP=tmpP;
      optX=x;
    end
  end

  % si en la parte inferior de la recursión
  % se tiene que el conjunto de elección estará vacío
  % si el peso actual supera al peso máximo, entonces
  % establecer esta elección en [0, 1]
  if (l>n)
    C= [];
  else
    if ((CurW+w (l)) <=M)
      C= [1 0];
    else
      C= [0];
    end
  end

  % para todos los elementos en el conjunto de elección
  % efectuar la recursividad
  for (i=1: length(C))
    x (l) =C (i);
    [optP, optX, r]=backtrack2(l+1,CurW+w(l)*x(l),x,w,p,optP,optX,M,r);
  end
```

A-3 Algoritmo backtrack3 (poda del árbol sobre la base función delimitación)

Esta implementación es, backtracking aplicado al problema de la mochila con una poda más sofisticada, haciendo uso de las funciones de delimitación.

```
function [optP,optX,r]=backtrack3(l,CurW,x,w,p,optP,optX,M,r, ...
                  sorted)

% invocación: [optP,optX,r]=backtrack3(l,CurW,x,w,p,optP,optX,M,r,sorted)
% parámetros: l: el nivel recursivo, inicializa en 1
```

```
%          curW: el peso actual, inicializa en 0
%             x: valor actual de x, inicializa en ceros
%             w: el vector de pesos
%             p: el vector de costos
%          optP: el beneficio óptimo, inicializa en 0
%          optX: el valor óptimo para X, inicializa en []
%             M: el máximo peso
%             r: número de llamadas a la función, iniciando en 1
% sorted: indica si se ordenan los datos de entrada, iniciando en  0
% Esta función implementa dar vuelta atrás con la poda sobre la base
% de la función de delimitación, mochila racional para resolver el
% problema de la mochila.
% derechos del autor Lars Aurdal / Riskshospitalet

  % si los datos no están ordenados, los clasifican
  if (sorted==0)
    ratio=p. /w;
    [ratio, index]=sort (ratio);
    index=fliplr (index);
    w=w (index);
    p=p (index);
    sorted=1;
  end

  % obtiene la longitud del vector
  n=length (w);

  % si en la parte inferior de la recursión
  if (l==(n+1))
    r=r+1;   % incrementar el número de llamadas

    % calcula el beneficio actual
    tmpP=0;
    for (i=1: n)
      tmpP=tmpP+p (i)*x (i);
    end

    % si el beneficio actual supera los costos máximos
    % a continuación, actualice el máximo costo y optX
    if (tmpP > optP)
      optP=tmpP;
      optX=x;
    end
  end

  % si en la parte inferior de la recursión
  % se tiene que el conjunto de elección estará vacío
  % si el peso actual lo supera el peso máximo, entonces
  % establecer esta elección a 0, de lo contrario
  % escoger la opción en [0, 1]
  if (l== (n+1))
    C= [];
  else
    if ((CurW+w (l))<=M)
      C= [1 0];
```

```
    else
      C= [0];
    end
  end

  % calcular la función de límite
  tmpP=0;
  for (i=1 :( l-1))
    tmpP=tmpP+p (i)*x (i);
  end
  B=tmpP+rknap (p (l: length (p)), w (l: length (w)), M-CurW);

  % Si la función de límite indica error
  % regrese, al conjunto de elección más largo.
  if (B<=optP)
    return;
  end

  for (i=1: length(C))
    x (l) =C (i);
    [optP, optX, r]=backtrack3(l+1,CurW+w(l)*x(l),x,w,p,optP,optX,M,r, ...
                 sorted);
  end
```

A-4 Algoritmo saks (recocido simulado)

```
function [optp,optX,E,optE,c]=saks(X,W,P,m,i_max,alpha,T_0)

% Invocación: [optp,optX,E,OptE,c]=saks(X,W,P,m,i_max,alpha,T_0)
% Parámetros:   X: valor inicial
%               W: el vector de pesos
%               P: el vector de costos
%               m: peso máximo
%           i_max: número máximo de iteraciones
%           alpha: parámetro de refrigeración (refrigeración geométrica)
%             T_0: temperatura inicial
% Esta función implementa el algoritmo de recocido simulado
% para resolver el problema de la mochila.
% derechos de autor Lars Aurdal / Rikshospitalet

% inicializar el número de iteraciones en uno
i=1;

% obtiene la longitud del vector de entrada X
n=length(X);

% inicialmente, el valor óptimo es igual al valor X
optX=X;

% inicialice el costo óptimo
optp=optX*P';

% inicialice la temperatura
```

```
T=T_0;

% inicialice el vector de recuento
c=zeros (1, i_max);

% siempre y cuando no se supera el número de iteraciones
while (i<i_max)

    % obtenga una solución legal de la vecindad,
    % ver los comentarios de la función hN
    % abajo
    Y=hN (n, X, W, m);

    % si conseguimos una solución....
    if (~isempty(Y))

        % actualizar costos temporales
        px=X*P';
        py=Y*P';

        % si el nuevo costo es mejor que el anterior, aceptar
        if (py>px)
            X=Y;

        % si el Nuevo costo es mejor, que el costo en general, actualizar
            if (py>optp)
                optp=py;
                optX=Y;
            end

            % de todos modos aceptar la nueva solución, con un cierto P
        else
            r=rand;
            if(r<exp ((py-px)/T))
                X=Y;
                c (i) =1;
            end
        end

       % actualizar los vectores de costos de mejor funcionamiento
        E (i) =px;
        optE (i) =optp;
    end

    % incrementar contador y disminuir la temperatura
    i=i+1;
    T=alpha*T;
end

% Función para calcular la vecindad
% y la búsqueda de un Nuevo candidato.
% Esta función encuentra todos los candidatos legales en
% la vecindad y luego elige candidatos aleatoriamente.
function Y=hN (n, X, W, m)
N=ones (1, n)'*X;
```

```
N=xor (N, eye (n));
currW=N*W';
index= (currW<=m);
N=N (index, :);
sz=size (N, 1);
if (sz>0)
    r1=floor (sz*rand) +1;
    Y=N (r1, :);
else
    Y= [];
end
```

A-5 Algoritmo hillks (colina inclinada)

```
function [optp, optX] =hillks (X_0, W, P, m, i_max)

% Invocación: [optP, optX] =hillks (X_0, W, P, M, i_max)
% Parámetros: X_0: solución inicial
%               W: el vector de pesos
%               P: el vector de costos
%               m: peso máximo
%           i_max: número máximo de iteraciones
% Esta función implementa el algoritmo de escalar la colina
% para resolver el problema de la mochila.
% derechos del autor Lars Aurdal/Rikshospitalet

  % comience por inicializar el número de iteraciones a cero
  i=0;

  % obtiene la longitud de entrada X
  n=length (X_0);

  % inicialmente, el valor óptimo es igual a X
  optX=X_0;

  % inicialice el costo óptimo
  optp=optX*P';

  % siempre y cuando no se supere el número de iteraciones
```

```
while (i<i_max)

  % Aquí generamos el entorno o vecindad de la
  % solución X. En primer lugar, hacer una matriz en la que cada fila
  % es igual al actual X, entonces xor con la matriz identidad
  % sirve para calcular una nueva matriz que contiene los vectores
  % vecinos.
  N=ones (1, n)'*optX;
  N=xor (N, eye (n));

  % Basándose en las soluciones contenidas en N,
  % calcular un vector de correspondientes
  % valores actuales de peso.
  currW=N*W';

  % Basándose en las soluciones contenidas en N,
  % calcular un vector de correspondientes
  % valores actuales de costos
  currP=N*P';

  % Mantenga el costo corriente máxima correspondiente de la solución actual
  currP=currP.*(currW<=m);
  [currp, index]=max (currP);

  % Actualizar si este Nuevo costo es mejor que el costo óptimo existente
  if (currp>=optp)
    optp=currp;
    optX=N (index, :);
  else
    break;
  end

  % Incrementar el contador de iteraciones
  i=i+1;
end
```

A-6 Algoritmo tabuks (búsqueda tabú)

```
function [optp, optX] =tabuks(X, W, P, m i_max, L)

% Invocación: [optp, optX] =tabuks(X, W, P, m, i_max, L)
% Parámetros:   X: valor inicial
%               W: el vector de pesos
%               P: el vector de costos
%               m: peso máximo
%           i_max: número máximo de iteraciones
%               L: Vida útil de la lista tabú
% Esta función implementa el algoritmo de búsqueda tabú
% para resolver el problema de la mochila.
% derechos de autor Lars Aurdal/Rikshospitalet

  % Comience por inicializar el número de iteraciones a cero
   i=0;

  % Obtiene la longitud del vector de entrada X
  n=length(X);

  % Inicialice la lista tabú en cero
   tL=zeros (1, n);

  % Inicialmente, el valor óptimo es igual al valor de X
  optX=X;

  % Inicialice el costo óptimo
  optp=optX*P';

  % Siga, siempre y cuando no se supere el número de iteraciones
  while (i<i_max)

    % Disminuir los valores distintos de cero en la lista tabú.
    tL=tL-(tL>0);

    % Aquí generamos el entorno o vecindad de la solución X.
    % En primer lugar, hacer una matriz en la que cada fila
    % es igual al actual X, entonces xor con la matriz identidad
    % sirve para calcular una nueva matriz que contiene los vectores
    % vecinos.
    N=ones (1, n)'*X;
    N=xor (N, eye (n));

    % Based on the solutions contained in N,
    % calcular un vector de correspondientes
    % valores actuales de costos.
    currW=N*W';

    % Basandose en las soluciones contenidas en N,
    % calcular un vector de correspondientes
    % current profit values.
    currP=N*P';
```

```
  % Mantenga el costo de máxima corriente correspondiente
  % a la solución legal.
  currP=currP.*(currW<=m).*(tL==0)';
  [currp, index]=max (currP);

  % Mantenga la mejor X de la vecindad u entorno.
  % Actualizar la lista tabú para hacer una transición
  % de nuevo a la anterior X imposible para un periodo.
  X=N (index, :);
  tL (index) =L;

  % Actualizar si este nuevo costo es mejor que él, costo existente
  if (currp>=optp)
    optp=currp;
    optX=X;
  end

  % incrementar el contador de iteraciones
   i=i+1;
end
```

Bibliografía.

[1] Abadie J., *Integer and Nonlinear Programing*, American Elsevier Publishing Co., 1970.

[2] Aldrich D. W., *"A Decomposition Approach to the Mixed Integer Programing Problem"* PH. D. Dissertation, School of Industrial Engineering, Purdue University, 1969.

[3] Aurdal, http://www.aurdalweb.com/heuristics.html [consulta: 25-09-2013].

[4] Backtracking, http://www.lcc.uma.es/~av/Libro/CAP6.pdf [consulta: 20-09-2013]

[5] Balas E., *"An Aditive Algorithm for Solving Linear Programs with Zero-One Variables"* Operations Research Vol. 22-1, 1974.

[6] Baugh C. R., Ibaraka T., y Muroga S., *"Results in Using Gomory's All Integer Algorithm to Desing Optimum Logic Networks"* Operations Research Vol. 19, pp 1090-1096, 1971.

[7] Bazaraa Monkhtar S., Javis, John J., *Programación lineal y flujo de redes*. Limusa. Quinta edición, 1996.

[8] Benichou M., Gauthier J. M., Girodet P., Hentges G., Ribieri G., y Vincent o., *"Experiments in Mixed-Integer Linear Programing*" Mathematical Programing Vol. 1, pp 76-94, 1971.

[9] Childress J. P., *"Five Petrochemical Industry Applications of Mixed Integer Programing"* Bonner and Moore Assoc. Inc. Houston, Texas, 1969.

[10] Davies R. E., Kendrick D. A., y Weitzman M., *"A Branch and Bound Algorithm for 0-1, Mixed Integer Programing Problems" Operations Research*, Vol. 19, pp 1036-1044, 1971.

[11] Elaine Rich, Kevin Knight, *Inteligencia Artificial*, segunda edición, 1994.

[12] Fred Glover, Manuel Laguna, *Tabu Search Kluwer Academic Publishers*, 1997.

[13] Gorry G. A., y Shapiro J.F., *"An Adaptative Group Theoretic Algorithm for Integer Programing Problems"* Management Science, Vol. 17, pp 285-306, 1971.

[14] Graves R. L. y Wolfe P., (editores), *recent Advances in Mathematical Programing* McGraw-Hill, 1963.

[15] Greenberg H., and Hegerich R. L., *"A Branch Search Algorithm for the Knapsack Problem"* Management Science, Vol. 16, pp 327-332, 1970.

[16] Kolesar P., *"A Branch and Bound Algorithm for the Knapsack Problem"* Management Science, Vol. 13, pp 723-735, 1967.

[17] Land A. H., y Doig A. C., *"An Automathic Method for Solving Discrete Programing Problems"* Econometrica, Vol. 28 pp. 497-520, 1960.

[18] Little J. D. C., Murty K. G., Sweeney D. W., y Karel C., *"An Algorithm for the Traveling Salesman Problem"* Operations Research, Vol. 11, pp 979-989, 1968.

[19] López Mayo Guillermo. *Curso de programación lineal,* Facultad de Ciencias Físico Matemáticas, Benemérita Universidad Autónoma de puebla, Noviembre, 2005.

[20] Manne A. S., *"A Mixed Integer Algorithm for Project Evaluation"* International Bank in Reconstruction and Development, Washington, D. C., 1971.

[21] Ochoa Rosso Felipe, *"Applications of Discrete Optimization Techniques to Capital Investment and Network Synthesis Problems"* Research Report, R 68-43, Massachusets Institute of Technology, Cambridge, Enero, 1968.

[22] Prawda Witenberg Juan. *Métodos y modelos de investigación de operaciones,* vol. I Modelos Determinísticos, 1995.

[23] Srinivasan A. V., *"An Investigation of some Computational Aspects of Integer Programing"* Journal of the Association of Computing Machinery, Vol. 12, pp 525-535, 1965.

[24] Tabu Search, http://sisbib.unmsm.edu.pe/cap3.pdf [consulta: 01-09-2013].

[25] Tomlin J. A., *"Branch and Bound Methods for Integer and Non-Convex Programing"* pp 437-450 en el libro de Abadie [1], 1970.

[26] Trauth C. A., y Woolsey R. E., "MESA. *A Heuristic Integer Linear Programing Technique"* Reporte SC-RR-68-229, Sandia, Labs. Albuquerque, New México, 1968.