

Solución al Problema de la Mochila usando Búsqueda Local

Marialicia Suárez

Sergio Teñan

Resúmen.

El problema de la mochila surge cuando hay una asignación de recursos con restricciones de costo. Cada ítem tiene un costo y un valor, se busca obtener el mayor valor posible con un presupuesto dado. El termino *Problema de la mochila* evoca la imagen de un viajero con una mochila de tamaño fijo que debe llenar solo con los ítem mas útiles. Este problema se clasifica como un problema de optimización combinatoria y, computacionalmente, se categoriza como NP-Completo [1]. Este artículo propone solucionar el problema planteado usando una función heurística de búsqueda local.

Palabras Clave:

Optimización Combinatoria, Metaheurística, Búsqueda Local, Iterativo

Abstract.

The Knapsack problem arises when there is an allocation of resources with cost constraints. Each item has a cost and a value, it seeks to obtain the highest value possible with a given budget. The term *Knapsack Problem* evokes the image of a traveler with a backpack fixed size that should be filled only with the most useful item. This problem is classified as a combinatorial optimization problem and, computationally, categorized as NP-Hard. This article proposes to solve the problem raised using a local search heuristic function.

Keywords:

Combinatorial Optimization, Metaheuristics, Local Search, Iterative

1. Introducción.

El problema de la mochila (KP por sus siglas en inglés, *Knapsack Problem*) es un problema que ha sido intensamente estudiado por muchos años. El interés en este problema se debe, por una parte a sus aplicaciones directas e indirectas, y por la otra, por ser un campo de prueba que permite ensayar la eficiencia de métodos de solución de búsqueda inteligente en problemas de optimización combinatoria.

Este artículo busca probar la eficiencia de la heurística de optimización combinatoria con búsqueda local utilizando las estrategias de pivoteo *random* y *best-improvement*.

2. Búsqueda Local.

Búsqueda local es un método determinístico y sin memoria, utilizado en muchos otros problemas de optimización. Se puede ver como un proceso iterativo que empieza en una solución y la perturba realizando modificaciones locales, buscando en su vecindad por una mejor solución. Si la encuentra, reemplaza su solución actual por la nueva y continua con el proceso, hasta que no se pueda mejorar la solución actual.

Algorithm 1: Búsqueda Local

Input: S_0 Solución inicial generada

Output: S Optimo local

```

1  $S = S_0$ 
2 while  $S$  no es optimo local do
3    $S' \in N_{(S)}$  con  $f_{(S')} < f_{(S)}$  (Solución mejor
   dentro de la vecindad)
4    $S \leftarrow S'$ 
5 return  $S$ 
```

El diseño de la vecindad es fundamental para el desempeño del algoritmo. Al hablar de vecindad nos referimos a un conjunto de soluciones cercanas a la solución actual. Para seleccionar cual solución escoger de esa vecindad, existe distintos acercamientos, siendo posible escoger entre el acercamiento de mejor mejora (*best improvement rule*), primer mejora (*first improvement rule*), aleatorio (*random rule*) entre otros.

2.1. Estrategias.

La estrategia de pivoteo, puede llegar a ser crucial para el desarrollo del algoritmo, pudiendo generar óptimos locales distintos según el caso, y es importante saber escoger cual estrategia es la mas adecuada para la función objetivo en cuestión. En la realización de este artículo, se usaron las reglas de pivoteo de *best-improvement* y *random*.

2.1.1. Mejor Mejora.

Bajo esta estrategia, se toma el S' perteneciente al conjunto de soluciones vecinas de S ($S' \in N(S)$) que genera la mejor mejora, es decir, del conjunto de todas las soluciones que superan a la solución actual, se toma el supremo, y se sustituye $S \leftarrow S'$ para continuar con la iteración.

2.1.2. Primer Mejora.

La estrategia de selección de primer mejora, comienza a recorrer la vecindad $N(S)$, comparando cada S' que toma con el S actual, al momento que consigue un S' mejor que la solución actual, lo toma, para de recorrer el vecindario y sustituye $S \leftarrow S'$

3. Problema de la Mochila.

El problema KP es un problema de optimización combinatoria de formulación sencilla, aunque su resolución es compleja, y que aparece, directamente o como un subproblema en una gran variedad de aplicaciones, incluyendo planificación de la producción, modelización financiera, muestreo estratificado, planificación de la capacidad de instalaciones, etc.

Además de sus potenciales aplicaciones, el KP es de particular interés por sus características combinatorias y su estructura sencilla, que lo vuelven un problema ideal para el diseño de métodos de búsqueda inteligente. Con el tratamiento del problema KP se pueden evaluar las ventajas y desventajas de estos algoritmos, sobretodo su precisión y rapidez.

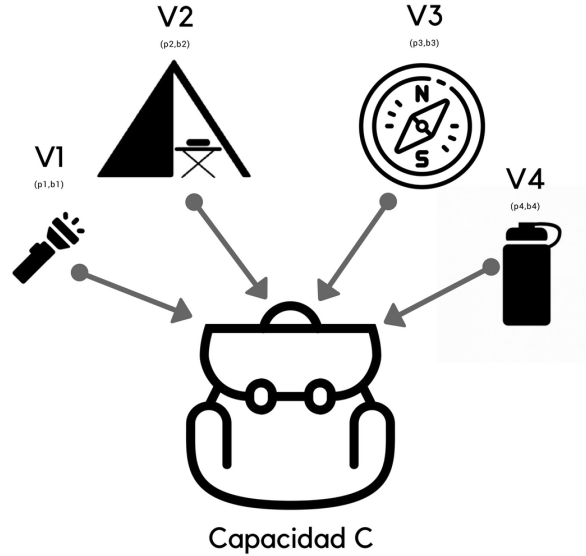
El KP es un problema combinatorio que es NP-Completo [2], que figura en la lista de 21 problemas NP-completos de Karp, por tanto es improbable que, en algún momento, pueda ser encontrado un algoritmo que pueda resolverlo en tiempo polinomial. A pesar de esto, no se cuenta como un problema fuertemente NP-Completo, por lo que puede ser resuelto en tiempo pseudo-polinomiales particularmente por programación dinámica. Para la realización de este artículo, se implementó una solución con programación dinámica, para considerar como casos óptimos, y realizar la comparación de resultados con la heurística en cuestión (LS).

3.1. Formulación.

Dado un conjunto de objetos $V = \{V_1, V_2, \dots, V_n\}$, donde cada V_i tiene un peso p_i y un beneficio b_i respectivamente, donde $i \in [1, n]$, y dado que se tiene un recipiente (Mochila) con capacidad C , el problema KP consiste en determinar la combinación de objetos de V que se agregan en la mochila, de manera que el beneficio obtenido sea el máximo posible, como se puede ver en la Figura 1, donde se tienen 4 items, es decir

$V = (p_1, b_1), (p_2, b_2), (p_3, b_3), (p_4, b_4)$ y la mochila tiene una capacidad total de C .

Figura 1: Problema de la Mochila. Como cargar óptimamente la mochila sin rebasar su capacidad máxima.



3.2. Función objetivo.

En el KP el objetivo es maximizar la suma de los beneficios, por lo que la función objetivo del problema se puede representar como

$$\text{Max} \sum_{i=1}^n b_i x_i$$

Esta función está sujeta a las restricciones de no negatividad y no nulidad además de la capacidad máxima, que no puede ser excedida por la sumatoria de los pesos depositados en la mochila.

$$\sum_{i=1}^n p_i x_i \leq c$$

$$x_i \in \{0, 1\}; i = 1, 2, \dots, n$$

La variable x_i toma el valor de 1 o 0 según el elemento V_i pertenece a la combinación dentro de la mochila o no respectivamente.

4. Búsqueda local para KP.

La idea de este estudio es comparar la eficiencia de los procedimientos de búsqueda local bajo las estrategias de *best-improvement* y *first-improvement* para resolver el problema KP. Para esta comparación se utilizaron instancias de pruebas generadas de manera aleatoria, y

se compararon los resultados de la heurística de LS, con los resultados obtenidos por la solución de programación dinámica implementada.

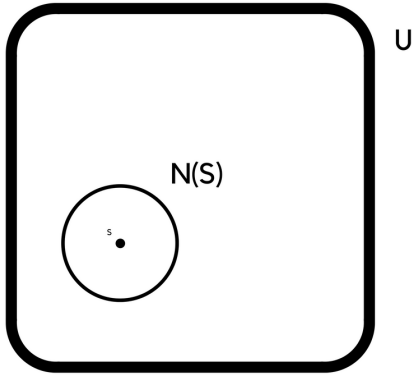
4.1. Estructura de vecindad.

Para decidir la forma en la cual se debe determinar el espacio de soluciones $N_{(S)}$ del problema que se aborda, debemos conocer el problema [3], sin olvidar que el objetivo es maximizar.

Comenzamos partiendo de una solución factible que cumpla con las restricciones del modelo matemático. Los elementos que conforman esta solución son aquellos cuyo peso no sobrepasa el peso soportado por la mochila. La solución inicial también son todos aquellos elementos cuya sumatoria de pesos es menor que la capacidad máxima. Por simplicidad se toma como $x_i = 0$ para $i = 1 \dots n$.

Para ilustrar la vecindad (Figura 2), se toma la solución $S = x_i; i = 1 \dots n$, y se consideran los $S' = x_i$ donde solo una de las x_i cambie su valor ($x_i \in \{0, 1\}$).

Figura 2: Dado un conjunto de soluciones U la vecindad $N_{(S)}$, indicada por el círculo, se define como los conjuntos S' que solo difieren en un elemento respecto a S (punto dentro de $N_{(S)}$).



Se observa que $S \in U$ y que $N \subset U$. Si se toma un punto $S' \in N_{(S)}$ que represente una mejora respecto a S , este se tomara como nueva solución actual. De este modo el metodo de LS va iterando paso a paso desde una solución inicial hacia una solución que mejore la anterior, que se evalúe en la función objetivo y que cumpla con todas las restricciones.

$$f(S') \leq f(S)$$

5. Resultados.

Para la implementación del algoritmo de programación dinámica se utilizó el lenguaje de programación Haskell, mientras que para los algoritmos de LS se utilizó el lenguaje de programación Python. Para los casos de uso se utilizaron dos *laptops*, para los casos en Haskell se utilizó un procesador Core i3 con 2Gb de RAM, y para los casos en Python se utilizó también un procesador i3 con 4Gb de memoria RAM. Luego de hacer 10 casos de prueba distintos, donde se varían la cantidad de ítems en V desde 10 hasta 50, se presentan los resultados, comparando los óptimos obtenidos para cada caso.

6. Discusión y Conclusiones.

Aquí concluyen.

Referencias

- [1] Papadimitriou, C. H., Steiglitz, K. Combinatorial Optimization: Algorithms and Complexity. *New York: Englewood Cliffs*, 1998.
- [2] M. R. GAREY Y D. S. JOHNSON, Computers and Intractability: A Guide to the Theory of NP-Completeness. *New York: W.H. Freeman*, 1979.
- [3] Joyanes, A. L., Zahonero, M. I. Programación en C: Metodología, algoritmos y estructura de datos. *México: McGraw-Hill*, 2000.

Apéndice