Estudio de Aplicación de Metaheurísticas al problema Maximum Diversity Problem (MDP)

Cristian Medina Manuel Pacheco

Abstract

El problema de máxima diversidad (Maximum Diversity Problem), consiste en elegir M elementos de un conjunto de N elementos tal que las suma de las distancias entre los elementos elegidos sea la máxima posible. Catalogado como un problema NP-Hard en 1993 por Kou, Glover y Dhit, el problema está formalmente estipulado bajo la siguiente fórmulas de función objetivo a maximizar y restricción de tamaño:

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} d_{ij} x_i x_j$$

$$\sum_{i=1}^{n} x_i < m$$

Donde n es la cardinalidad de elementos del conjunto, m la cantidad de elementos elegidos, d_{ij} es la distancia entre el elemento i y el elemento j, x_i es una función booleana que toma el valor de 1 si el elemento i fue elegido, 0 si no fue elegido. Estas definiciones son las utilizadas en menciones posteriores en caso de omitirse una definición en el momento.

Las aplicaciones de este problema se pueden encontrar en estudios genéticos, sistemas ecológicos, controles de inmigración, tratamientos médicos, entre otros. Ha sido un caso de estudio para la aplicación de heurísticas y metaheurísticas, particularmente para la búsqueda local. Fue catalogado como un problema NP-Hard en 1993 por Kou, Glover y Dhit al reducir el problema de clique a MDP.

El presente artículo presenta un estudio sencillo de la aplicación de metaheurísticas para la resolución de este problema. Se realiza un estudio comparativo de las técnicas de Local Search (LS), Iterated Local Search (ILS), Tabu Search (TS), Genetic Algorithm (GA) y Scatter Search (SS) en las intancias del problema en los sets SOM, GKD y MDG de MDPLIB de optsicom.es.

1. Trabajos Previos

El problema de máxima diversidad ha sido estudiado aplicando múltiples heurísticas y metahaurísticas. En general todas las heurísticas estudiadas presentan buenos resultados, sin embargo, se destacan las de búsqueda local al poseer resultados de mejor calidad para la mayor parte de los casos de prueba.

Se destaca en particular el Basic VNS por Brimberg [1]. Este consiste en una estructura de datos eficiente que representa los conjuntos de datos actualizados, del cual genera aleatoriamente una solución parcial a la cual le aplica una búsqueda local intercambiando los elementos de su solución con los elementos no incluidos. En un segundo lugar, se encuentra el VNS implementado por Aringhieri y Cordone [2], el cual consiste en una construcción de una solución con un método greedy y un posterior procesamiento cambiando k elementos de la solución y aplicando una búsqueda tabú simple sobre esta.

Para casos de prueba de mayor tamaño, se destaca las búsquedas en Tabú. Un ejemplo de esta es la presentada por Palubeckis [5], una búsqueda Tabú iterativa que alterna entre búsqueda tabú y técnicas de perturbación. Cuando la búsqueda Tabú encuentra un resultado mejor al inicial, utiliza una búsqueda local a la nueva solución. Su método de perturbación es una selección de elementos aleatorios del conjunto para viajar por el espacio de soluciones. Para casos pequeños este método se desempeña bien, pero no es el mejor.

Cabe destacar también los estrategias GRASP desarrolladas por Silva et [3], donde se construye una solución parcial inicial y se mejora de forma iterativa; y la propuesta por Duarte y Martí [4], que parten de la heurística propuesta por Glover de C2 y D2, aplicándole su propia metodología GRASP.

2. Representación de la solución

Para la representación de una solución del problema se utilizan dos alternativas. Para los algoritmos LS, ILS, TS y SS se optó por una representación compacta. Sea el conjunto de elementos de tamaño n y la solución de tamaño m, se opta por representar la solución como un vector de tamaño m donde cada componente contiene la etiqueta del elemento escogido. Para GA se utiliza una representación completa: un vector de tamaño n donde cada componente está permanentemente asociada a un

elemento del problema e indica si pertenece o no a la solución (i.e. un x_i como en la descripción del problema).

3. Solución Inicial

Dependiendo de la metaheurística a utilizar se usan métodos de construcción de las soluciones iniciales de forma diferente. La primera es la clásica solución aleatoria: se seleccionan los elementos que pertenecen a la solución al azar. La segunda sigue un principio de intuición que denominamos *Potencial* de un elemento definida de la siguiente forma:

$$P(i) = \sum_{j=1}^{n} d_{ij}$$

De manera informal, el *Potencial* de un elemento indica cuánto aporta a la función objetivo si se tomasen en cuenta todas las conexiones con todos los demás elementos. Por tanto, elegimos los primeros m nodos que tengan mayor potencial. Definimos a esta estrategia como solución inicial greedy.

En la mayoría de los problemas trabajados, las soluciones aleatorias rondan generalmente entre 50 % y 75 % del valor de las mejores soluciones encontradas en la librería (soluciones referencia), dependiendo del tamaño del problema, siendo las peores las de problemas más grandes. La solución greedy supera el 95 % y tiende a llegar tan alto como 98 % de la solución referencia.

En la mayoría de los problemas se intenta utilizar la solución greedy sobre la aleatoria. Para LS, ILS y TS se utiliza greedy. Para las metaheurísticas poblacionales GA y SS, se inicia con soluciones aleatorias.

4. Local Search (LS)

El primer algoritmo implementado consiste en Búsqueda Local. Durante la búsqueda, el objetivo es mejorar una solución existente modificando aquellos elementos que lleven la solución a acercarse a un óptimo (al menos de forma local).

La implementación de LS esta basada en el uso de *k–opt*, con 4 características en las cuales hacer énfasis:

- Se busca mejorar los m componentes de la solución.
- Los vecinos de cada componente son definidos en base al tamaño n del problema.
- Cada componente es reemplazada por el mejor candidato encontrado
- La búsqueda por componente se detiene si ha pasado un numero de iteraciones sin conseguir una solución candidata que mejore la solución actual.

4.1. Algoritmo de Local Search

El algoritmo utilizado es equivalente al aquí mostrado, abstrayendo las particularidades de implementación y optimización del algoritmo. Esto aplica a todos los algoritmos mostrados en el presente documento.

```
1  S ← Solucion del problema
2  S_i ← elemento i de la solucion
3  U ← elementos no escogidos
4  P ← elementos del problema
5  begin
6      S ← SolucionInicial()
7      foreach i ∈ |S| de derecha a izquierda do
8          U' ← NH_SIZE elementos aleatorios de U
9          foreach u ∈ U' do
10             S' ← S con S_i reemplazado por u
11             if valor(S') > valor(S) then
12                 S ← S'
13             if pasan MAX_REPEATS iteraciones
               sin cambiar S then
14                 break
```

Para NH_SIZE y MAX_REPEATS se entonaron los valores de $\frac{|P|-|S|}{5}$ y 30 respectivamente.

4.2. Resultados de Local Search

La aplicación de LS a una solución dada, en general logra mejorar la solución y acercarla hacia el óptimo (en general, local). Quizá uno de los resultados más importantes en una revisión individual de LS, es cómo se comporta con diferentes soluciones. En la figura 1, se hacen 15 pruebas sobre 5 problemas del set mediano iniciando con una solución aleatoria (r) y una solución greedy (g). El comportamiento observado es el esperado: local search mejora localmente una solución dada; con una solución greedy se obtienen mejores resultados pero sucesivas aplicaciones de LS a una solución arbitraria podrían igualmente obtenerlos.

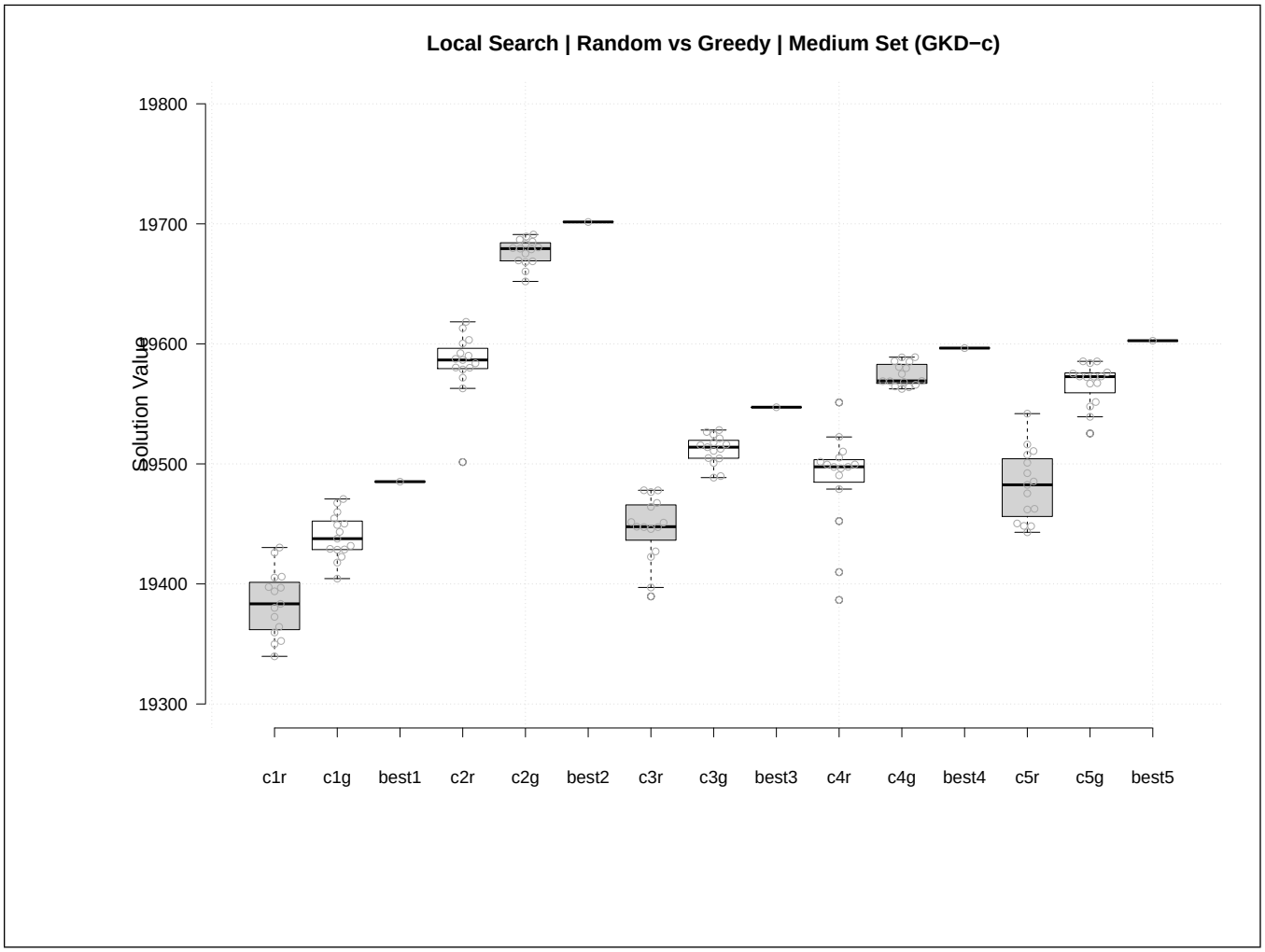

Figura 1: *Comparación de soluciones de LS utilizando soluciones random y greedy en problemas de tamaño mediano.*

5. Iterated Local Search (ILS)

En la búsqueda ILS, se intenta resolver el problema principal que aqueja a la búsqueda local: puede quedarse estancado en soluciones óptimas locales, sin llegar al óptimo global. Esto puede resolverse aplicando búsqueda local sobre diferentes soluciones iniciales, mas conservando de cierta forma aquello que las búsquedas anteriores pudieran aportar para hallar la solución óptima.

5.1. Operador de perturbación de ILS

Para cambiar las soluciones, realizamos una perturbación en base a un tamaño k, en la que se eligen k elementos escogidos en la solución y se reemplazan con elementos no escogidos en la solución actual.

5.2. Condiciones de control de ILS

En la implementación existen 3 condiciones que controlan la forma en que se realizan las búsquedas:

- Intentos de k: cuantas busquedas con el mismo tamaño k se permiten sin mejora antes de incrementarlo.
- Incremento de k: en cuanto se incrementa el tamaño de la perturbación y el límite de este incremento.
- Máximo de Iteraciones: cuantas iteraciones totales se permiten realizar durante la ejecución completa del algoritmo.

5.3. Algoritmo de ILS

```
1  S ← Solucion del problema
2  k ← tamaño de la perturbacion
3  begin
4      S ← SolucionInicial()
5      k ← INIT_SIZE
6      k_tries ← 0
7      for i = 1 → MAX_ITERATIONS do
8          S' ← shake(S,k)
9          S'' ← localSearch(S')
10         if value(S'') > value(S)) then
11             S ← S''
12             k ← INIT_SIZE
13             continue
14         increment k_tries
15         if k_tries > MAX_K_TRIES then
16             increment k
17             k_tries ← 0
18             continue
19         if k > |S| then
20             break
```

6. Genetic Algorithm (GA)

El algoritmo genético consiste en una simulación genética de las soluciones, donde se posee una población donde cada individuo representa una instancia de solución y se aplican simulaciones de selección, cruce y mutación genéticas a modo de imitación del proceso natural de los seres vivos. Se implementó un algoritmo genético generacional para la resolución de este problema: las nuevas poblaciones reemplazan a las anteriores, manteniendo el mismo tamaño en la población durante todo el algoritmo.

6.1. Representacion del Cromosoma de GA

Dada las características del problema, la representación utilizada en las demás metaheurísticas era poco útil/eficiente para la implementación de un operador de cruce en GA. En esta metaheurística, se utiliza la representación completa, descrita en la sección *Representación de la solución"*.

6.2. Población Inicial

Se utiliza una población inicial de tamaño fijo y cada elemento es construido de forma aleatoria.

6.3. Operador de Selección de GA

Para la selección de los individuos a ser utilizados en el cruce, se utilizó roulette-wheel sampling con elitismo simple: se seleccionan individuos al azar con tendencia a los individuos con mejor "fitness". El fitness es directamente proporcional al valor de la función objetivo. Con respecto al elitismo simple, no se permite elegir al mejor individuo de la población.

6.4. Operador de Cruce de GA

Dadas las características del problema El operador de cruce implementado es un operador personalizado, de 2 padres a 2 hijos, descrito informalmente de la siguiente forma: *en las componentes en que los padres coinciden, los hijos coinciden. En las componentes que solo 1 padre lo posee, se reparten alternativamente entre los 2 hijos.* Adicionalmente, se tiene la condición de que los hijos reemplazan a sus padres si poseen mejor fitness.

6.5. Operador de Mutación de GA

Utilizamos un operador de mutación bastante simple y débil: se intercambia un elemento presente en la solución por uno que no este presente en la solución. Sin embargo, la mutación se usa con una muy alta probabilidad: al final de cada generación, 1 de cada MUTATION_SIZE elementos (seleccionados al azar) son mutados. Esta alta probabilidad de mutación resultó en mejores resultados para GA en los problemas medianos.

6.6. Algoritmo de GA

```
1  P ← Poblacion de invididuos
2  C ← Individuos Elegidos
3  S_i ← Solucion i de un conjunto de soluciones
4  begin
5      P ← PoblacionInicial(POP_SIZE)
6      for i = 1 → N_GENERATIONS do
7          C ← Seleccion(P)
8          foreach S_i, S_{i+1} ∈ C do
9              S'_i, S'_{i+1} ← Cruce(S_i, S_{i+1})
10             if fitness(S'_i) > fitness(S_i) then
11                 replace(S'_i,S_i,P)
12             if fitness(S'_i + 1) > fitness(S_i + 1) then
13                 replace(S'_{i+1},S_{i+1},P)
14         Mutacion(P,MUTATION_SIZE);
```

Para los valores de POP_SIZE, N_GENERATIONS y MAX_REPEATS se entonaron los valores de 100, 200 y

4 respectivamente.

7. Resultados

7.1. Computador de Pruebas

Las pruebas fueron realizas bajo un equipo con las siguientes propiedades:

- Sistema Operativo: Ubuntu 15.04 de 64 bits.
- Kernel de SO: 3.19.0-21
- Procesador: Intel Core i7-4700MQ
- RAM: 8 GB RAM DDR3

7.2. Casos de Prueba

Para las pruebas se utilizaron los casos de prueba de la librería MDPLIB:

- SOM-b: 5 problemas de n entre 100 y 200 y m entre 10 y 40.
- GKD-c: 20 problemas de n 500 y m 50.
- MDG-b/c: 5 problemas de n entre 2000 y 300 y m entre 200 y 600.

Referencias

[1] Brimberg, J., N. Mladenovic, D. Urosevic and E. Ngai. (2009). Variable neighborhood search for the heaviest k-subgraph. *Computers & Operations Research*, 36(11): 2885-2891.

[2] R. Aringhieri and R. Cordone. Better and faster solutions for the maximum diversity problem. *Technical report, Universit degli Studi di Milano, Polo Didattico e di Ricerca di Crema*, 2006.

[3] G.C. Silva, M.R.Q. Andrade, L.S. Ochi, S.L. Martins, and A. Plastino. New heuristics for the maximum diversity problem. *Journal of Heuristics*, 13(4):315–336, 2007.

[4] R. Aringhieri, R. Cordone, and Y. Melzani. Tabu search vs. grasp for the maximum diversity problem. *A Quarterly Journal of Operations Research*, 6(1):45–60, 2008.

[5] G. Palubeckis. Iterated tabu search for the maximum diversity problem. *Applied Mathematics and Computation*, 189:371383, 2007.