Técnicas de Soft Computing. Optimización y algoritmos bioinspirados

Diseño de metaheurísticas para problemas de optimización

Autor:
Jesús Enrique Cartas Rascón

Profesor:
Manuel Lozano Márquez

Índice general

1. Maximum Diversity Problem

1.1. Definición del problema

El Maximum Diversity Problem (MDP) Problem consiste en, dado un conjunto de elementos, obtener el subconjunto en el que la diversidad de los elementos sea máxima. Esto es, dado un conjunto N con cardinalidad $|N| = n$, obtener un subconjunto $M \subset N$, con cardinalidad $|M| = m$ y $m < n$ en el que se maximice la diversidad de sus elementos.

La función de diversidad se puede definir tal que:

$$MD(x) = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n} D_{ij} x_i x_j \tag{1.1}$$

donde la solución es un número, que determina la diversidad de los elementos elegidos.

D es una matriz de distancias que asocia la distancia de dos elementos del conjunto N, de forma que D_{ij} determina la distancia entre el elemento i y el elemento j en el conjunto N.

x es una solución posible, que se compone de n elementos donde $x_i \in \{0, 1\}$, es decir, un vector binario. Este vector indica los índices de los elementos del conjunto N que estamos tomando como posible combinación. De esta forma, la cardinalidad de M, se calcula como:

$$m = \sum_{i=0}^{n} x_i \tag{1.2}$$

lo que nos dice el número de 1 que hay en el vector.

El problema se resume así a calcular qué combinación x ofrece una diversidad mayor. Atendiendo a la ecuación 1.1 vemos que el cálculo se sustenta sobre la formación del vector x. Al multiplicar D_{ij} por x_i y x_j, si alguno de ellos fuera 0, el producto se anula. Solo nos quedamos con aquellos índices en los que ambos sean 1, y el resultado del producto es la distancia entre estos dos elementos (que no necesariamente es conmutativa). Esto nos deja con un vector de distancias cuya suma debemos maximizar.

1.2. Revisión bibliográfica

Haciendo una revisión de los diferentes artículos relacionados con nuestro problema en SCOPUS, encontramos varias propuestas.

En *GRASP and tabu search for the generalized dispersion problem* [1], proponen una solución basada en GRASP [2] combinada con una búsqueda tabú [3].

Una solución basada en *iterated greedy* [4] fue propuesta por profesores de la universidad de Granada. [5].

Finalmente, se propuso una heurística a este problema basada en la técnica *path relinking*, combinada, una vez más, con GRASP. [6]

Como vemos, el problema es muy difícil de resolver (NP-duro, concretamente [7]). Es muy necesario utilizar técnicas de optimización que resuman el espacio de búsqueda en alternativas más asequibles. Es común el uso de GRASP, ya que, generalmente, da buenos resultados. Se proponen en las siguientes secciones algunas propuestas a nivel conceptual con objeto de resolver el problema en un tiempo razonable.

1.3. Operadores de vecindad y búsqueda local

Para definir un operador de vecindad, debemos tener clara cuál es la forma de nuestra solución y, más importante, cual *no* es la forma de nuestra solución.

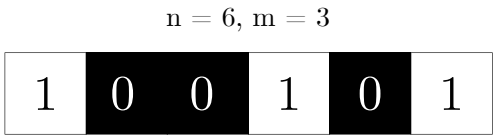

Figura 1.1: Representación gráfica de una solución posible, con 6 elementos y un subconjunto m de 3.

1.3.1. Operador de intercambio

En la figura 1.1 podemos ver un ejemplo de una solución candidata posible, dado que el superconjunto N tenga cardinalidad 6 y el subconjunto M, 3.

De tener un número distinto de 1, no se respetaría la restricción de $m = 3$ por lo que el espacio de búsqueda se acota exclusivamente a esta solución y todas las permutaciones posibles. El espacio de búsqueda tiene un tamaño de

$$\binom{n}{m} = \frac{n!}{(n-m)!\ m!} \tag{1.3}$$

ya que solo necesitamos las combinaciones que contengan m número de unos.

Dado esto, podemos presentar muchas alternativas para saltar de una solución a otra. Una de las más comunes e intuitivas sería intercambiar las posiciones de dos elementos.

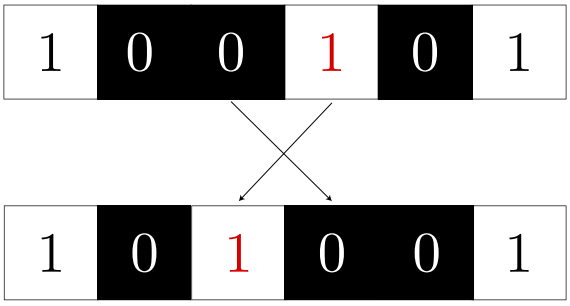

Figura 1.2: Posible definición de operador

En la figura 1.2 podemos ver cómo intercambiamos las posiciones para obtener una nueva solución. Este operador es muy simple y, en la mayoría de los casos, ofrece una nueva solución candidata viable; es decir, que respeta las restricciones inicialmente dadas.

Hay ocasiones en las que esto no ocurrirá. Por ejemplo, en la figura 1.3 vemos que para dada una posible solución, al aplicar el operador, se nos devuelve la misma exacta solución. Esto es un problema que deberemos tener en cuenta.

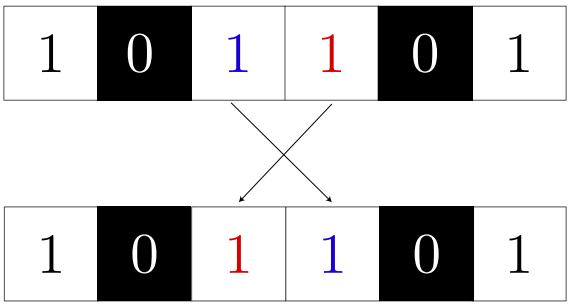

Figura 1.3: Al haber varios 1 consecutivos, el operador se aplica pero no cambia la solución.

1.3.2. Operador cilíndrico

Otra alternativa es interpretar la solución como un vector circular. Para generar una nueva solución, simplemente *rotamos* la solución hacia una dirección predeterminada.

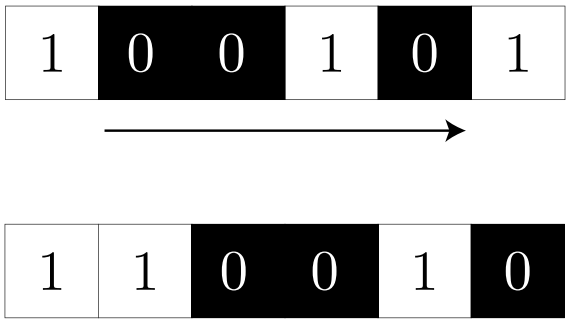

Figura 1.4: Demostración del operador circular. Todos los elementos se mueven, en este caso, un paso a la derecha. El último se sitúa el primero.

La ventaja de este operador es que siempre nos dará soluciones distintas y factibles. Una posible desventaja es su escaso grado de exploración. Dada una solución con n elementos, solo tenemos $n - 1$ elementos para explorar, lo que puede limitarnos bastante.

1.3.3. Búsqueda local

Determinados los operadores de vecindad, podemos definir un procedimiento de búsqueda local.

Este procedimiento es muy rudimentario, pero funcionaría relativamente bien. Dado el operador de vecindad, generamos una solución pseudo-aleatoria o completamente aleatoria en el espacio de soluciones. Después, buscamos todos sus vecinos, comparamos sus niveles de diversidad y, si lo superan, nos quedamos con esa solución. Esto volverá a repetirse, generando los vecinos de la nueva solución propuesta. Este algoritmo no irá hacia atrás, es decir, no tomará la solución antes tomada, ya que la solución anterior no tendrá una diversidad mayor. De haber sido así, no hubiera sido elegida en primer lugar.

Se usan unas variables de estado como *found* y *stop* que nos asisten en el proceso de la condición de parada. Si hemos pasado por todos los vecinos y ninguno supera a la actual solución considerada, decimos que no hemos encontrado nada. Si no hemos encontrado nada, paramos.

```
⊕ ← Definir operador de vecindad;
s ← Generar solución pseudo-aleatoria inicial;
D_s ← Diversidad(s);
while not stop do
    V_{s,⊕} ← Cálculo de todos los vecinos directos;
    found ← False;
    foreach v ∈ V_{s,⊕} do
        D_v ← Diversidad(v);
        if D_v > D_s then
            s ← v;
            D_s ← D_v;
            found ← True;
        end
    end
    if not found then
        stop = True;
    end
end
return s, D_s
```

Algorithm 1: Búsqueda local

Esta técnica también se denomina **hill climbing**, ya que, buscamos de entre todos los vecinos posibles y siempre vamos a aquel que más beneficio nos aporte. Este algoritmo tiene el problema de quedarse estancado en óptimos locales, lo cual se soluciona aleatorizando ligeramente el proceso, con diferentes procedimientos. No obstante, como primera propuesta, es válido.

1.4. Metaheurísticas voraces

En esta sección propondremos algunas heurísticas basadas en el heurística voraz o *greedy*.

1.4.1. Greedy

El algoritmo voraz o greedy toma la mejor solución nada más encontrarla y continúa a partir de ahí. De ahí el nombre *voraz*, ya que trata de tomar la mejor solución que encuentra lo antes posible.

Este procedimiento funciona de una manera muy similar a la búsqueda local, con la clave diferencia de que la búsqueda local considera todos los pasos posibles y toma el mejor, mientras que el algoritmo voraz toma el primero que mejore su actual solución.

```
⊕ ← Definir operador de vecindad;
s ← Generar solución pseudo-aleatoria inicial;
D_s ← Diversidad(s);
while not stop do
    V_{s,⊕} ← Cálculo de todos los vecinos directos;
    found ← False;
    foreach v ∈ V_{s,⊕} do
        D_v ← Diversidad(v);
        if D_v > D_s then
            s ← v;
            D_s ← D_v;
            found ← True;
            break;
        end
    end
    if not found then
        stop = True;
    end
end
return s, D_s
```

Algorithm 2: Metaheurística voraz

Como vemos, la diferencia está en que, en el ciclo interno, nada más encontrar una solución mejor, lo terminamos y pasamos a evaluar esta nueva mejor solución. Esto acelera el proceso si el espacio de vecindad es grande, además de que tiene un factor exploratorio mucho más grande que la búsqueda local. Sin embargo, puede que no llegue a converger, o que tarde mucho en hacerlo.

1.4.2. Semi Greedy

Los algoritmos semi-greedy [8] no eligen el mejor directamente, sino que eligen un vecino de forma aleatoria de los c mejores que haya en su vecindad. Por ejemplo, si tenemos un espacio de vecindad de 10, y $c = 3$, elegiremos aleatoriamente un vecino del subconjunto de los 3 mejores.

```
c ← determinar c;
⊕ ← Definir operador de vecindad;
s ← Generar solución pseudo-aleatoria inicial;
D_s ← Diversidad(s);
while not stop do
    V_{s,⊕} ← Cálculo de todos los vecinos directos;
    D_V ← Diversidad(V_{s,⊕});
    Top_c ← BestNeighbours(D_V, c);
    found ← False;
    while Top_c is not empty do
        r ← RandomChoice(Top_c);
        D_r ← Diversidad(r);
        if D_r > D_s then
            s ← r;
            D_s ← D_r;
            found ← True;
        else
            Top_c ← Remove(Top_c, r);
        end
    end
    if not found then
        stop = True;
    end
end
return s, D_s
```

Algorithm 3: Metaheurística semi-voraz

El algoritmo considera los c mejores candidatos y elige uno aleatoriamente. Si el candidato no es mejor que la solución actual, se elimina de la lista y se repite el proceso. Si no hubiera un mejor candidato, la lista estaría vacía, lo que daría paso a terminar el algoritmo.

Aclarar en la sentencia $D_V \leftarrow$ Diversidad($V_{s,\oplus}$) que el cálculo de vecindad se ejecuta sobre todos los elementos del vector, es decir, todos los vecinos. Esto es fácilmente recreable en lenguajes de programación con técnicas de vectorización y de programación funcional.

1.4.3. Iterated Greedy

La familia de heurísticas iterated greedy [4] funcionan utilizando mecanismos de destrucción y reconstrucción de soluciones. La solución propuesta, s pasa por una fase de destrucción, que nos deja con una solución parcial incompleta s_p, para pasar a otra fase de reconstrucción en la que se genera una nueva solución completa, s'. A partir de aquí, podemos definir un criterio de aceptación que elija solo soluciones que mejoren, o introducir algún factor aleatorio que aumente el grado de exploración.

De esta forma, podemos definir una propuesta muy general del algoritmo tal que:

$s_0 \leftarrow$ Generar solución pseudo-aleatoria inicial;
while *not criterio de aceptación* **do**
    $s_p \leftarrow$ Destrucción(s^*);
    s' $\leftarrow$ Construcción(s_p);
    CriterioAceptación(s^*, s');
end

Algorithm 4: Metaheurística voraz-iterativa

Es un esquema muy general porque la definición de cada función de Destrucción y Construcción puede ser diferentes, así como la del criterio de aceptación. Pero en general, la metaheurística siempre seguirá una estructura similar a la propuesta.

1.5. Algoritmos genéticos: poblaciones, cromosomas, cruces y mutaciones

En esta sección definiremos los parámetros necesarios para encontrar una solución al problema mediante algoritmos genéticos.

Los algoritmos genéticos son una técnica de exploración que simula el proceso evolutivo de la naturaleza. Dada una serie de individuos de una población, estos individuos serán tanto mejores cuanto la solución que proponen se acerque más a la óptima. Dada esta regla, se forman generaciones, en las que las peores soluciones *mueren*, y las mejores se cruzan (simulando reproducción). Esto favorece una exploración del espacio de búsqueda muy interesante. Para añadir factores aleatorios, al final de cada generación, estos individuos pueden mutar.

Todos estos hiperparámetros serán controlados y determinarán la calidad de nuestra solución.

1.5.1. Cromosomas

Una de las partes más importantes de la implementación de un algoritmo genético es la definición de la población y de sus componentes; en este caso, cromosomas.

La idea radica en definir un conjunto de *individuos* o *genes*, cada uno compuesto de unas características. Estas características se mutarán y cruzarán con otros miembros de la población, lo que ofrece una exploración del espacio de búsqueda muy interesante.

Para ello, debemos definir muy bien cómo es cada miembro de esta población. Usualmente, son vectores que albergan las características de nuestro problema y que suponen una solución del mismo. En problemas más complejos, esta es la parte más difícil, la de encontrar una representación tal y como la que definimos.

En nuestro caso, por suerte es trivial. La disposición de la solución tal y como se definió en la primera sección corresponde perfectamente con una representación *genética*.

La figura 1.1 nos ofrece una representación de lo que queremos.

1.5.2. Operadores genéticos

Obtenida la representación de los cromosomas, ahora definiremos los operadores de cruce y mutación, que nos permiten explorar el espacio de búsqueda.

Cruce

El operador de cruce es simple: dados dos individuos, toma una sección del primero e inclúyela en el segundo, formándose un nuevo individuo que tiene características de los dos *padres*.

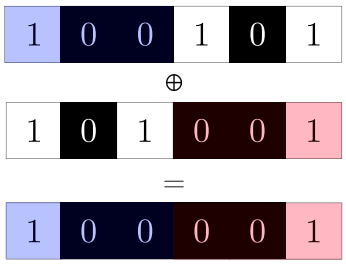

Figura 1.5: Representación gráfica del operador de cruce clásico

De forma más técnica, podemos representarlo como se ilustra en la figura 1.5. Se escoge un punto aleatorio donde cortamos, y el hijo se compondrá de la sección del *padre* anterior al corte y la sección posterior de la *madre*.

Sin embargo, este método tan simple rompe las reglas de nuestro problema. Como vemos, la cardinalidad de la solución m no es 3, como la de sus padres, sino 2. Deberemos así ingeniar un método alternativo que trate de cumplir las restricciones pero conservando el factor exploratorio de los operadores de cruce.

Se propone aquí el siguiente operador de cruce que trata de solventar este problema.

1. Para empezar, se toma una sección de tamaño aleatorio del *padre*. Esta sección se copia directamente en el *hijo*.

2. Se calcula el número de unos que esta sección nos ha dado, para ver cuántos nos faltan.

3. Finalmente, se recorre el vector del otro progenitor, asegurando que encontramos tantos unos como hagan falta. Para suplir por el posible exceso, se eliminan todos los ceros de esta sección que sean necesarios para que la solución sea válida.

Se ilustra un ejemplo gráficamente, paso por paso en la figura 1.6

Este método no nos libra, sin embargo, de obtener un hijo que sea exactamente igual que alguno de sus padres. Esto se puede solucionar de dos maneras: o bien repetir el proceso otra vez o mutar el resultado.

Mutación

De la misma manera, el operador de mutación está estrechamente relacionado con la representación y las restricciones de nuestra solución. Mutar consiste en alterar de forma aleatoria alguna de las componentes de nuestros individuos, añadiendo un factor exploratorio adicional.

Esto puede llevarnos de nuevo al problema de antes: generar soluciones no válidas. Para ello, diseñaremos nuestro operador con ello en mente.

Inicialización de la población

Una vez definidos los individuos de la población y las interacciones entre ellos, lo único que queda es el desarrollo de la *simulación*. Para ello, debemos generar un número g de individuos iniciales.

La forma más sencilla es generarlos aleatoriamente. En nuestro caso, como sabemos, esto puede llevar a soluciones incorrectas. Una posible manera de inicializar una población en nuestro contexto particular es la siguiente.

Dado un vector de partida que puede ser provisto por el usuario o generado sistemáticamente dados los parámetros (por ejemplo, para $n = 6$ y $m = 3$ generamos el vector [111000]), barajamos ese vector. Para generar la muestra n, barajamos la muestra $n - 1$. Esto asegura que solo obtenemos permutaciones válidas.

Se puede considerar el hecho de que se repitan algunos individuos tras las permutaciones, caso bastante improbable en números grandes de n y m, pero, por supuesto, no 0. Aún así, puede ser interesante tener alguna instancia repetida ya que, aunque albergue la misma información, no seguirá el mismo camino que su análogo al interactuar con otros individuos, formando generaciones diferentes.

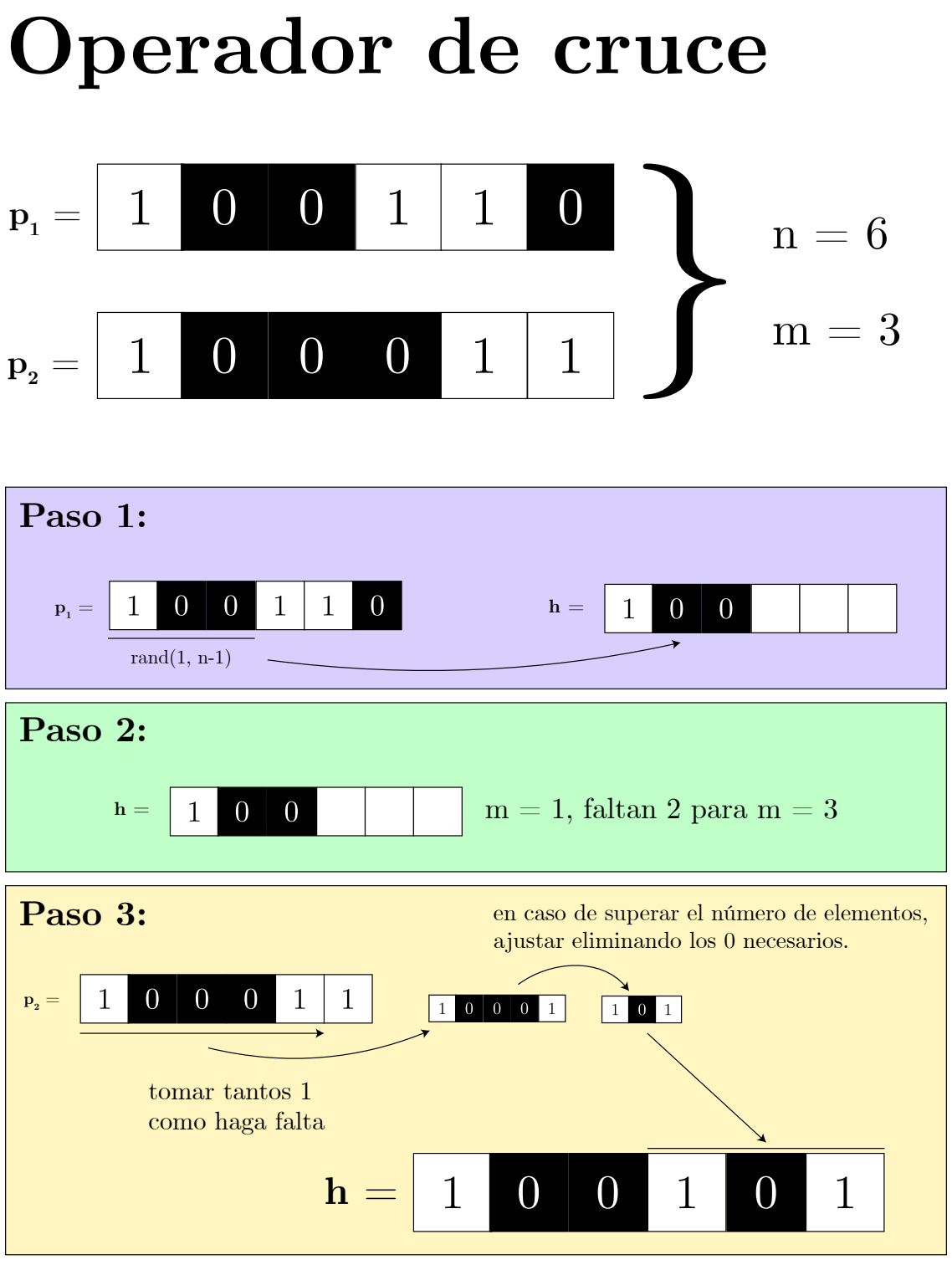

Figura 1.6: Ilustración del operador de cruce sugerido para cumplir con las restricciones del problema

2. Multidimensional two-way number partitioning

2.1. Definición del problema

El problema de Multidimensional two-way number partitioning (M2NP) consiste en minimizar una suma de vectores obteniendo una partición binaria del conjunto principal. Dado S, nuestro conjunto principal, que se compone de vectores de tamaño d, obtenemos una partición binaria de S, S_1 y S_2, en la que las particiones son disjuntas, $S_1 \cap S_2 = \emptyset$ y su unión es el superconjunto S, $S_1 \cup S_2 = S$.

Una vez obtenidos los subconjuntos posibles, sumamos coordenada a coordenada cada uno de los vectores de cada conjunto, lo que nos resume el subconjunto en una tupla de tamaño d. Obtenidas las dos sumas, obtenemos el valor absoluto de la diferencia de los resultados, cordenada a coordenada también, como ilustramos en la figura 2.1. Las operaciones se ven reflejadas con los marcos de colores: la operación en rojo suma todos los vectores por columnas y nos devuelve los vectores suma. Dados los vectores suma, hacemos la operación del valor absoluto de la diferencia.

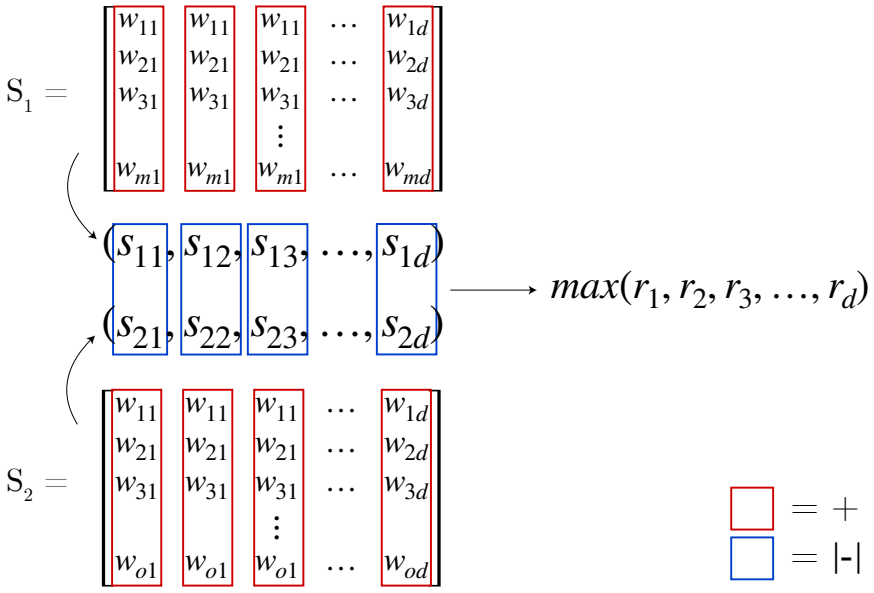

Figura 2.1: Ilustración gráfica de la función de evaluación del M2NP

Esto nos deja con una tupla de tamaño d, de la que obtendremos el máximo. La idea es minimizar este valor.

2.2. Revisión bibliográfica

Haciendo una revisión de los diferentes artículos relacionados con nuestro problemaen SCOPUS, encontramos varias propuestas.

Alexander Faria utiliza una mixed-integer linear programming técnica de programación mixta para resolver el problema en [9]. Por otro lado, Valentino Santucci se decantó por una estrategia evolutiva en [10]. Finalmente, vemos otra vez el uso de GRASP con path relinking en [11], técnica que parece dar buenos resultados.

2.3. Operadores de vecindad y búsqueda local

Como establecimos en el capítulo anterior, es clave definir una representación útil de una solución viable a nuestro problema para poder definir operadores de vecindad y búsqueda local, así como cualquier heurística.

En nuestro caso, tenemos un vector de vectores inicialmente. Sin tener en cuenta que cada elemento es un vector, la idea es descomponer el conjunto original en dos. Al ser una descomposición binaria, podemos representar una solución con números binarios de igual forma. En este caso, la solución se interpreta de manera diferente: el número tendrá tantos dígitos como el conjunto original tenga elementos, n. Agruparemos todos los elementos con un 0 en el grupo de S_1, y los que tengan un 1, en S_2.

Figura 2.2: Igual que antes, podemos representar la solución con un vector de dígitos binarios.

Desde el punto de vista de restricciones, esta solución cumple con todas desde el principio. Las únicas restricciones es que los subconjuntos deben ser disjuntos, lo que se cumple naturalmente. La unión de los conjuntos nos devuelve S también. No hay una restricción de un máximo de unos como antes, y todas las posibles permutaciones son viables.

Incluso la partición de todo ceros o todo unos (que, en realidad, proporcionan el mismo resultado) sería válida, ya que restaríamos coordenada a coordenada con $w = (0, 0, 0, ...)$. El valor absoluto de la resta final es el que determina que el resultado sea el mismo.

2.4. Metaheurísticas voraces

Definida la función de evaluación de las soluciones y una representación de las soluciones, podemos pasar a diseñar las diferentes metaheurísticas.

Definidas ya en la sección 1.4, estas heurísticas resuelven nuestro problema pero atendiendo a las restricciones del anterior. Simplificándolas para que no se tenga que hacer el cálculo de una solución posible, ya que todas en el espacio de búsqueda lo son, obtendríamos las soluciones para este problema.

2.4.1. Greedy

```
⊕ ← Definir operador de vecindad;
s ← Generar solución pseudo-aleatoria inicial;
D_s ← FunciónEvaluación(s);
while not stop do
    V_{s,⊕} ← Cálculo de todos los vecinos directos;
    found ← False;
    foreach v ∈ V_{s,⊕} do
        D_v ← Diversidad(v);
        if D_v > D_s then
            s ← v;
            D_s ← D_v;
            found ← True;
            break;
        end
    end
    if not found then
        stop = True;
    end
end
return s, D_s
```

Algorithm 5: Metaheurística voraz

La diferencia clave entre este diseño y el del problema anterior se hallaría en la función del cálculo de todos los vecinos directos, en la que, en nuestro caso, viene simplemente dado por el operador de vecindad. En el problema anterior, debíamos solo tomar aquellos vecinos que respetaran las restricciones del problema.

2.4.2. Semi Greedy

En la misma torna, el algoritmo semi greedy luciría igual:

```
c ← determinar c;
⊕ ← Definir operador de vecindad;
s ← Generar solución pseudo-aleatoria inicial;
D_s ← Diversidad(s);
while not stop do
    V_{s,⊕} ← Cálculo de todos los vecinos directos;
    D_V ← Diversidad(V_{s,⊕});
    Top_c ← BestNeighbours(D_V, c);
    found ← False;
    while Top_c is not empty do
        r ← RandomChoice(Top_c);
        D_r ← Diversidad(r);
        if D_r > D_s then
            s ← r;
            D_s ← D_r;
            found ← True;
        else
            Top_c ← Remove(Top_c, r);
        end
    end
    if not found then
        stop = True;
    end
end
return s, D_s
```

Algorithm 6: Metaheurística semi-voraz

2.4.3. Iterated Greedy

Y, finalmente, del mismo modo, la heurística del iterated greedy también luciría igual. Todo se resume al concepto de que no tenemos ningún tipo de restricción en nuestro problema, lo que realmente simplifica su diseño.

```
s_0 ← Generar solución pseudo-aleatoria inicial;
while not criterio de aceptación do
    s_p ← Destrucción(s*);
    s' ← Construcción(s_p);
    CriterioAceptación(s*, s');
end
```

Algorithm 7: Metaheurística voraz-iterativa

2.5. Algoritmos genéticos: poblaciones, cromosomas, cruces y mutaciones

Definiremos en esta última sección una propuesta para los algoritmos genéticos.

2.5.1. Cromosomas

Como ya sabemos, la solución se representa como un vector de ceros y unos, de longitud igual a la cardinalidad del conjunto S inicial. Esto define un esquema muy similar al del anterior problema.

2.5.2. Operadores genéticos

Definiremos ahora los operadores genéticos del M2NP.

2.5.3. Cruce

En cuanto al cruce, podemos efectuar cualquier tipo de operación. La clásica, definida en la sección 1.5.2 y en la figura 1.5 sería una muy buena opción. Tomamos la mitad del *padre* y la mitad de la *madre* para formar al nuevo individuo de la siguiente iteración.

Hay otros operadores de cruce muy interesantes que pueden explorar el espacio de búsqueda de una forma alternativa.

La primera propuesta es igual que el algoritmo clásico, pero tomando más de un punto de corte, usualmente, dos. Dados los puntos de corte, los dos fragmentos del *padre* se copian en el hijo conservando la posición, y el fragmento de la *madre* que queda dentro de los dos puntos se copia de igual forma. Por supuesto, se puede hacer justo al revés, obteniéndose dos soluciones por operación. Se ilustra esta operación en la figura 2.3.

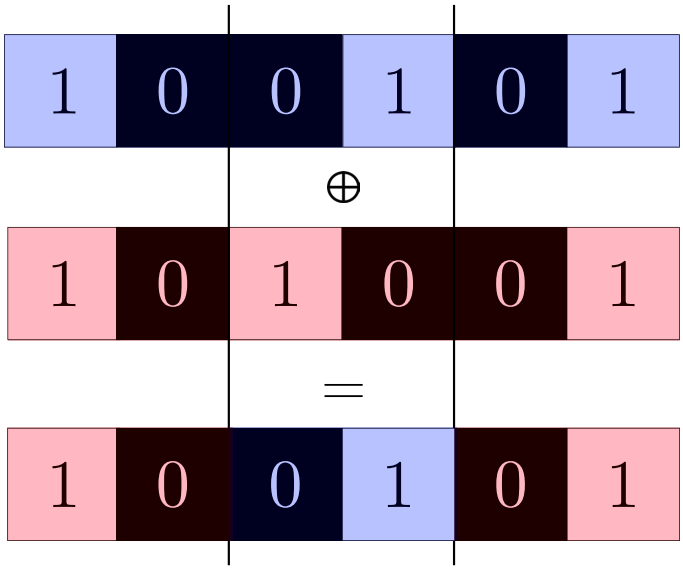

Figura 2.3: Operador de cruce con dos puntos de corte. La posición de los cortes también se puede aleatorizar para ofrecer un nivel de exploración más alto.

2.5.4. Mutación

En cuanto a la mutación, la operación puede ser tan fácil como escoger un índice aleatoriamente y cambiar su valor. Como son binarios, solo hay que elegir qué índice cambiar y alternarlo. En otras soluciones con representaciones de enteros habría también que elegir qué numero poner en el índice elegido.

2.5.5. Inicialización de la población

De cara a inicializar la población, generaríamos de forma completamente aleatoria el número de muestras que deseemos lanzar en la simulación, dado que cada solución se compone de n dígitos siendo $n = |S|$.

Bibliografía

[1] Anna Martínez-Gavara, Teresa Corberán, and Rafael Martí. Grasp and tabu search for the generalized dispersion problem. *Expert Systems with Applications*, 173:114703, 2021.

[2] Mauricio G. C. Resende and Celso C. Ribeiro. *Greedy Randomized Adaptive Search Procedures*, pages 219–249. Springer US, Boston, MA, 2003.

[3] Fred Glover and Manuel Laguna. *Tabu Search*, pages 2093–2229. Springer US, Boston, MA, 1998.

[4] Thomas Stützle and Rubén Ruiz. *Iterated Greedy*, pages 547–577. Springer International Publishing, Cham, 2018.

[5] M. Lozano, D. Molina, and C. García-Martínez. Iterated greedy for the maximum diversity problem. *European Journal of Operational Research*, 214(1):31–38, 2011.

[6] M. R. Q. De Andrade, P. M. F. De Andrade, S. L. Martins, and A. Plastino. Grasp with path-relinking for the maximum diversity problem. In *Lecture Notes in Computer Science*, volume 3503, pages 558–569, 2005.

[7] Ching-Chung Kuo, Fred Glover, and Krishna S. Dhir. Analyzing and modeling the maximum diversity problem by zero-one programming. *Decision Sciences*, 24(6):1171–1185, 1993.

[8] J.Pirie Hart and Andrew W. Shogan. Semi-greedy heuristics: An empirical study. *Operations Research Letters*, 6(3):107–114, 1987.

[9] A.F. Faria, S.R. de Souza, and E.M. de Sá. A mixed-integer linear programming model to solve the multidimensional multi-way number partitioning problem. *Computers and Operations Research*, 127, 2021.

[10] V. Santucci, M. Baioletti, G. Di Bari, and A. Milani. A binary algebraic differential evolution for the multidimensional two-way number partitioning problem. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11452 LNCS:17–32, 2019.

[11] F.J. Rodriguez, F. Glover, C. García-Martínez, R. Martí, and M. Lozano. Grasp with exterior path-relinking and restricted local search for the multidimensional two-way number partitioning problem. *Computers and Operations Research*, 78:243–254, 2017.