UNIVERSIDAD DE MURCIA
FACULTAD DE INFORMÁTICA
INGENIERÍA EN INFORMÁTICA

TÉCNICAS HEURÍSTICAS Y METAHEURÍSTICAS PARA EL PROBLEMA DE LA MÁXIMA DIVERSIDAD (MAXIMUM DIVERSITY PROBLEM (MDP))

PROYECTO FIN DE CARRERA

Autor:

MIGUEL ÁNGEL FRANCO GÁLVEZ
Alumno de Ingeniería Superior Informática
Facultad de Informática
Universidad de Murcia
mafg1@um.es

Director:

DOMINGO GIMÉNEZ CÁNOVAS
Grupo de Computación Científica y Programación Paralela
Departamento de Informática y Sistemas
Universidad de Murcia
domingo@um.es

Codirector:

JOSÉ MATÍAS CUTILLAS LOZANO
Colaborador del Grupo de Computación Científica y Programación Paralela
Departamento de Informática y Sistemas
Universidad de Murcia
josematias.cutillas@um.es

Murcia, Septiembre de 2015

Resumen

En este proyecto se trata el problema de la "Máxima diversidad" (Maximum Diversity Problem (*MDP*)), uno de los problemas pertenecientes a los problemas del tipo de optimización combinatoria tan útilies en la actualidad. Se sabe que este tipo de problemas son de la máxima complejidad a la hora de resolverlos y que, por ello, se suelen utilizar heurísticas/metaheurísticas para su resolución.

Por ello se ha realizado un algoritmo, al que hemos llamado "GRASP_M", que intenta aportar algo útil a la resolución del problema *MDP*. Este algoritmo se ha desarrollado sin tener en cuenta, a priori, las técnicas/metaheurísticas utilizadas hasta el momento para su resolución. Teniendo en cuenta los resultados obtenidos, se cree se ha creado un algoritmo eficiente en la resolución del problema *MDP*, y pudiera ser que se haya aportado alguna idea al campo de estudio de resolución de este problema.

También se ha analizado y realizado pruebas sobre un enfoque completamente distinto para abordar los problemas de optimización combinatoria, como lo es el problema *MDP*. Este enfoque es el de un "Esquema Metaheurístico Parametrizado" (*EMP*) donde se propone un algoritmo generalizado cuyo funcionamiento se basa en unos parámetros, llamados "Parámetros Transicionales", que permite la emulación de diversos tipos de metaheurísticas (*GRASP*, *GA* (*Genetic Algoritms*) y *SS* (*Scatter Search*)), así como hibridaciones (incluso nuevas metaheurísticas) de estas, y su aplicación a varios tipos de problemas de tipo de optimización combinatoria con un mínimo de cambios en la implementación del mismo.

En los dos casos se analiza una metodología para la determinación de valores satisfactorios para los parámetros incluidos en los esquemas algorítmicos utilizados.

Como conclusiones sobre las pruebas realizadas sobre los distintos algoritmos, se obtienen resultados interesantes sobre el rendimiento de "GRASP_M", y alguna posible aportación, así como algunos objetivos a mejorar de *EMP*, cuyo planteamiento es de mayor nivel y no está directamente enfocado a la resolución eficiente del *MDP* .

Índice general

Índice de tablas

Listados de código

Capítulo 1

Introducción

En una sociedad actual cada vez más compleja, conectada e interrelacionada, surgen nuevos retos para el análisis y estructuración de la misma que permitan una adecuada organización tal que se favorezca un uso adecuado y óptimo de su recursos, tanto materiales como de los individuos que la forman.

En este escenario, los métodos que resuelven el Problema de la Máxima Diversidad (MDP en adelante) se revelan como una herramienta muy útil para encontrar una solución efectiva a multitud de estas necesidades donde la optimización y la diversidad son factores clave a tener en cuenta.

En este capítulo introductorio se introducirá brevemente al lector al problema MDP, se expondrán la motivación y objetivos de este trabajo, y se mostrará la estructura del documento.

1.1. Problema MDP

El Problema de la Máxima Diversidad, más conocido por *MDP* (*Maximum Diversity Problem*) por sus siglas en inglés, es un problema ampliamente tratado en la literatura y aplicable a multitud de situaciones de la vida actual y cotidiana, como se comentará más adelante.

La formulación básica matemática del problema se introdujo en [27] como:

$$Maximizar \quad \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} d_{ij} x_i x_j \tag{1.1}$$

$$Condicionado\ a \quad \sum_{i=1}^{n} x_i = m \tag{1.2}$$

$$con \quad x_i = \{0,1\} \quad 1 \leq i \leq n \tag{1.3}$$

Se trata de encontrar un conjunto M con m elementos de entre otro conjunto N de n elementos, $M \subseteq N$, tal que la diversidad de M sea máxima. La diversidad entre dos elementos i, j pertenecientes a N se representa como d_{ij} y existen multitud de criterios propuestos para calcularla dependiendo de la situación concreta a la que se aplique el problema MDP.

La opción más sencilla es considerar la distancia euclidiana $d(x_i, x_j)$ entre dos elementos x_i y x_j pertenecientes a N, representada por d_{ij} como:

$$d_{ij} = \sqrt{(x_i - x_j)^2} \equiv |x_i - x_j| \tag{1.4}$$

para un sólo valor de x_i y x_j, que se puede generalizar si consideramos que cada elemento de N tiene p atributos como:

$$d_{ij} = \sqrt{\sum_{k=1}^{p} (x_{ik} - x_{jk})^2} \tag{1.5}$$

Como es evidente, de este modo las distancias son siempre ≥ 0.

Hay quien considera (véase el análisis en [39]) que este tipo de medida de la diversidad es muy restrictivo y no adecuado para algunas aplicaciones del problema. Así, por ejemplo, se proponen otro tipo de medidas de la diversidad, como puede ser *la medida de similitud del coseno* en [28] y *la medida de similitud en grupos de solucionadores de problemas* [23], donde se pueden obtener valores tanto positivos como negativos, y que dicen ser más adecuadas para su aplicación a grupos de personas.

Las distancias tratadas en este trabajo serán obtenidas del conjunto de datos de referencia utilizado [33] y no son calculadas por el algoritmo desarrollado, es decir, no nos centramos en el cálculo de las distancias en sí, sino que procedemos a intentar solucionar el problema con unas distancias ya dadas. No debería suponer un problema adaptar los algoritmos para tratar con cualquier otro tipo de medida de diversidad (distancias).

Es conveniente indicar que este problema ha sido tratado en la literatura existente con distintos nombres y enfoques, lo que dificulta la búsqueda de información sobre el mismo. Una muestra de estos nombres distintos podría ser la especificada en el siguiente listado tomado de [30]:

- p-dispersion, or p-dispersion-sum (Erkut 1990; Kincaid and Yellin 1993)
- edge-weighted, or remote clique (Macambira and Souza 2000; Chandra and Hall- dorsson 2001)
- maximum edge-weighted subgraph (Macambira 2002)
- dense k-subgraph (Feige et al. 2001)
- p-defense, or p-defense-sum (Moon and Chaudhry 1984; Cappanera 1999)
- equitable dispersion (Prokopyev et al. 2009)
- maxisum, or max-avg, dispersion (Kuby 1987; Ravi et al. 1994)

Se ha demostrado que el problema *MDP* es *NP-duro* (*NP-hard*) [27], es decir, es un problema de alta complejidad computacional. Esto supone que el tiempo de resolución crece exponencialmente con el tamaño del problema, lo que hace que no sea factible la resolución exacta para todas las instancias del problema, sobre todo para las de mayor tamaño tanto del conjunto N como del M. Es por esto que, históricamente, se han desarrollado y

aplicado técnicas heurísticas y metaheurísticas para intentar obtener soluciones suficientemente buenas, pero no necesariamente exactas (óptimas), que permitan tratar este tipo de problemas en un tiempo computacional aceptable.

La complejidad de este problema viene dada por el número de soluciones factibles que alberga el problema. El número de soluciones factibles viene dado por la expresión combinatoria $\binom{n}{m}$, donde n es la cardinalidad del conjunto de elementos disponibles y m es la cardinalidad del subconjunto de N que queremos maximizar. Así, por ejemplo, para un problema con 10 elementos de los que buscamos maximizar la diversidad de un subconjunto de 3 elementos de entre esos 10 tendríamos un número de soluciones factibles $\binom{10}{3} = 120$, que es fácilmente abordable por un algoritmo exacto. Pero digamos que tenemos un conjunto con 500 elementos y queremos encontrar un subconjunto de 50 elementos que maximice la diversidad. Ahora se tendrían un número de soluciones factibles $\binom{500}{50} = 2314422827984300469017756871661048812545657819062792522329327913362690 \simeq 10^{69,364}$. Vemos como para tamaños no demasiado grandes o irrealistas la utilización de métodos exactos para la resolución del problema no es factible.

Para ver la importancia que tiene este problema, así como sus variantes en cuanto a distintas funciones objetivo: *Max-Min*, *Max-MinSum*, *Min-Diff* y *Max-Mean* [39], veamos algunos campos en los que puede aplicarse:

- Localización de unidades logísticas para un buen rendimiento y no solapación de recursos [20].
- Ubicación de instalaciones peligrosas, contaminantes o estratégicas para evitar riesgos [10].
- Composición de jurados públicos en juicios [29].
- En los procesos de generación de nuevos medicamentos en la industria farmaceútica [35].
- En la creación de políticas de inmigración [27].
- Plantas de cría de animales, problemas sociales, preservación ecológica, control de polución, diseño de productos, inversión de capitales, diseño de curriculum, ingeniería genética [20, 27].
- Y otros muchos campos potenciales de aplicación que no se mencionan en esta lista o aún están por ver.

1.2. Motivación y objetivos

Vista la especificación del problema, su complejidad, en función del tamaño del mismo, y la importancia que tiene en la resolución de situaciones de la vida real, se intentará aportar un granito de arena al estado del arte actual de los algoritmos y métodos

existentes que abordan el problema *MDP*, a la vez que se analiza el estado del arte actual de los métodos de resolución de este problema.

Se intentará crear un algoritmo heurístico para solucionar el problema *MDP* sin tener demasiado en cuenta lo que se ha hecho y se está haciendo hasta el momento en este campo.

No se pretende realizar un algoritmo rompedor que mejore a todos los demás, lo que sería muy presuntuoso, sino intentar abordar el problema desde un punto de vista personal y basado, a partes iguales, en la intuición y la experimentación. Posiblemente habrá muchas coincidencias con algoritmos heurísticos ya presentados en otros trabajos (este es un campo de estudio que suscita mucho interés recientemente) pero, como ya se ha dicho, no se busca el refinamiento o modificación de ninguna de las anteriores soluciones.

Una vez presentado el algoritmo heurístico se diseñarán y realizarán los experimentos necesarios para comprobar el buen o no tan buen funcionamiento del mismo en términos de bondad ("fitness") y tiempos de resolución de *MDP*.

Tras la presentación del algoritmo heurístico creado se realizará un estudio de algunos de los métodos de resolución más utilizados para *MDP* con la ayuda de un algoritmo que aporta un esquema parametrizado metaheurístico [1], se analizarán los resultados, tanto en relación a la bondad de las soluciones como a los tiempos de ejecución, se compararán resultados frente a la heurística aportada y se obtendrán las conclusiones correspondientes.

1.3. Estructura del documento

En este capítulo 1 se ha realizado una breve introducción al Problema de Máxima Diversidad (*MDP*), ya que es un problema ampliamente tratado en la literatura, y se define la motivación y objetivos de este trabajo.

En el capítulo 2 se introducirá al lector en el estado del arte de las técnicas actuales postuladas y utilizadas para la resolución del *MDP*.

En el capítulo 3 se analizará la aplicación de heurísticas al *MDP*, se diseñará un algoritmo basado en heurísticas y se realizará un análisis de su bondad en cuanto a su aplicación a la resolución del *MDP*.

En el capítulo 4 se analizará la aplicación de metaheurísticas utilizando un esquema metaheurístico parametrizado (*EMP*) al problema *MDP*, y se confrontarán los resultados con los obtenidos por la heurística aportada.

Por último, en el capítulo 5 se mostrarán las conclusiones obtenidas en este trabajo así como posibles líneas de trabajo futuras.

Se incluyen dos apéndices, tras la sección de referencias bibliográficas, en los que se muestran pequeños manuales de usuario para los dos algortimos tratados en la experimentación de este trabajo.

Capítulo 2

Métodos de resolución aplicables a MDP

A continuación veremos algunos de los métodos exactos, heurísticos y metaheurísticos que se han utilizado a lo largo del tiempo que se lleva tratando la resolución del problema *MDP*. La razón de dividir los métodos en estas tres categorias radica en los distintos enfoques con los que afrontan la resolución de este problema de optimización combinatoria.

En la categoría de **métodos exactos** se incluyen los métodos que, en principio, explorarían todo el espacio de soluciones del problema para encontrar la solución óptima del mismo (no un óptimo local) si se ejecuta completamente el método. Se han propuesto formas de intentar disminuir el tamaño del espacio de soluciones explorado de tal forma que el tiempo de obtención de la solución disminuya. Dentro de estos métodos podemos citar el *Backtracking* y el *Branch and Bound*.

Los **métodos heurísticos** son aquellos métodos simples e intuitivos que buscan encontrar una buena solución al problema aplicado en un tiempo de ejecución factible. No pretenden explorar todo el espacio de soluciones (al igual que los métodos metaheurísticos) sino que, ya sea aleatoriamente o guiados por cierta inteligencia básica, esperan encontrar una solución buena (no necesariamente la óptima del problema) en un tiempo razonable. Ejemplos de estos métodos son el *Método de Construcción de Ghosh*, el *Método de Construcción de Glover* o la *Búsqueda Local de la Mejor Mejora* (*Best Improvement Local Search*).

Los **métodos metaheurísticos**, como su propio nombre indica ("Meta-" prefijo procedente del griego y que significa "más allá", "a un nivel superior"), se situan conceptualmente por encima de los heurísticos en el sentido de que guían el diseño de estos. Introducen mayor nivel de inteligencia al proceso heurístico y, por lo tanto, pretenden la obtención de mejores resultados. Ejemplos de estos métodos son *GRASP*, *Scatter Search*, *Variable Neighborhood Search*, *Algoritmos Genéticos*, etc.

2.1. Métodos exactos

Ya se ha explicado que la complejidad del problema impide que se pueda resolver de manera óptima para tamaños de problema grandes. Aun así, se han probado e investigado distintos métodos de resolución exactos del problema *MDP*, así como otros problemas de optimización combinatoria, en distintas variantes y aproximaciones que merece la pena tener en cuenta.

Se ha intentado optimizar la búsqueda de una solución mediante *Backtracking*, que explora todo el espacio de soluciones factibles (hay variantes en las que se intenta evitar esto), y mediante el algoritmo de *Ramificación y Acotación* (*Branch and Bound* en inglés), más apropiado para problemas de optimización.

Los algoritmos *Branch and Bound* intentan eliminar de la búsqueda las ramas del árbol de exploración que se sabe que no van a obtener una mejor solución a la mejor ya existente. La decisión de *Podar* (eliminar de la exploración una parte del árbol de soluciones, *"Prune"* en inglés) se realiza en base a unas *cotas* (*Bounds* en inglés) que se calculan durante la exploración y que sirven para indicarnos si vale la pena o no continuar por el camino que se lleva. En caso de no satisfacer los criterios para seguir por la rama de exploración se desecha (poda) y se continúa con la exploración por otra rama prometedora aún no explorada del árbol.

Es fundamental el buen cálculo de estas cotas para maximizar la eficiencia y corrección del algoritmo. También puede influir el orden de exploración del árbol de soluciones, tanto si se explora en profundidad como en anchura, así como el orden de los nodos del árbol a explorar. Estos factores pueden ser muy dependientes del problema tratado.

Aun con todas las posibles mejoras a los algoritmos exactos estos no pueden tratar, en un tiempo razonable, los problemas a partir de un determinado tamaño.

Se pueden consultar ejemplos de optimización de este tipo de algoritmos en [34] o en [4], donde se propone un enfoque interesante consistente en comenzar recorriendo el árbol de soluciones con unas cotas inicialmente restrictivas e ir afinando y relajando estas cotas, en pasos sucesivos, a la vez que se vuelve a recorrer el árbol de soluciones. No se consigue una mejora en cuanto a tiempos en la exploración global pero se sugiere como posible método para conseguir la generación temprana de resultados satisfactorios que se podrían incluir en la fase de inicialización de otras técnicas heurísticas. El método se llama *VWB* (*Variable Width Backtracking*).

Hay hibridaciones de métodos tanto metaheurísticos como heurísticos con métodos exactos que pueden consultarse en [38] donde se analizan las distintas formas de hibridación entre estos métodos, así como varios ejemplos aplicados a problemas de tipo combinatorio como el que nos ocupa.

2.2. Heurísticas

Veamos primero una posible definición de método o algoritmo heurístico dada en [9]:

> "Un método heurístico es un procedimiento para resolver un problema de optimización mediante una aproximación intuitiva, en la que la estructura del problema se utiliza de forma inteligente para obtener una buena solución"

De este modo podemos considerar un método heurístico como un método basado en la intuición, basada en la estructura del problema, guiada de forma inteligente (con un cierto criterio) para obtener una solución lo suficientemente buena, y no necesariamente la óptima, al problema tratado. Este tipo de métodos no suelen ser muy complejos y buscan, principalmente, la eficiencia resultados vs. tiempo. Por ejemplo, estos métodos no suelen utilizar ningún tipo de memoria para guíar la búsqueda de soluciones.

Las principales características deseables para un método heurístico son:

- **Eficiencia**. Un coste de computación razonable para conseguir una buena solución.
- **Bondad**. Las soluciones obtenidas por el método no deben alejarse en promedio de la solución óptima del problema.
- **Robustez**. Debe haber una baja probabilidad de obtener una mala solución.

Este tipo de métodos podrían dividirse en tres tipos según su enfoque al problema aplicado:

- **Métodos constructivos**. Tratan de ir construyendo una solución desde cero, o más de una si son multiarranque, hasta alcanzar una solución lo suficientemente buena.
- **Métodos de búsqueda local**. Dada una solución inicial intentan mejorarla (en cuanto al objetivo del problema) mediante la realización de cambios (movimientos) a la misma.
- **Métodos híbridos** o **combinados**. Realizan una conjunción de los dos métodos anteriores.

Ejemplos de métodos constructivos son *ErkC* [11], *GhoC* [14], *C2* y *D2* [20] o *STA* [37].

La idea de la búsqueda local fue introducida en los métodos constructivos *ErkC* y *GhoC*. Basándose en este idea se han implementado métodos más generales como *BLS* (Best Improvement Search) [11, 14] o *I_LS* (Improved Local Search) [8]. La diferencia principal de los distintos métodos de búsqueda local radica en la estrategia de la selección de los cambios (movimientos) que permitan obtener una solución mejorada.

La hibridación o conjunción de los dos métodos anteriores supone aplicar inicialmente una fase de construcción para obtener una solución inicial y, posteriormente, aplicar una búsqueda local para mejorarla. Se puede realizar una sola iteración o realizar multiples iteraciones, es decir, ejecutar el proceso un número de iteraciones o tiempo predeterminado.

Aunque los métodos heurísticos estén guiados por cierta inteligencia para llegar a buenas soluciones también se puede aplicar cierto nivel de aleatoriedad para evitar caer en mínimos locales y expandir la exploración del espacio de soluciones.

Puede consultarse más información sobre estos métodos heurísticos en [30].

Al ser este tipo de métodos simples, sin memoria del proceso de búsqueda y sin la aplicación de características avanzadas introducidas por las metaheurísticas, por sí solos no suelen obtener los mejores resultados en la obtención de soluciones de los problemas de optimización combinatoria como el *MDP*. Sin embargo, han proporcionado mecanismos básicos utilizados en posteriores formulaciones metaheurísticas, como puede ser la búsqueda local.

2.3. Metaheurísticas

Una posible definición de procedimiento *Metaheurístico* es la dada en [32]:

> "Los procedimientos metaheurísticos son una clase de métodos aproximados que están diseñados para resolver problemas difíciles de optimización combinatoria, en los que los heurísticos clásicos no son efectivos. Los Metaheurísticos proporcionan un marco general para crear nuevos algoritmos híbridos combinando diferentes conceptos derivados de la inteligencia artificial, la evolución biológica y los mecanismos estadísticos."

En concreto, los *métodos metaheurísticos* se sitúan conceptualmente por encima de los métodos heurísticos en el sentido de que guían el diseño de estos, es decir, establecen un marco donde se indica cómo y dónde aplicar métodos heurísticos junto con otras directrices que conforman un método sistemático para la resolución de un problema de una manera efectiva siguiendo un determinado planteamiento basado en observaciones obtenidas, ya sea de la naturaleza (evolución biológica, comportamiento de grupos, etc.) o la teoría en otros campos de estudio (inteligencia artificial, estadística, etc.).

Han sido propuestas multitud de metaheurísticas e hibridaciones de las mismas, algunas de las cuales veremos a continuación en más detalle.

2.3.1. GRASP (Greedy Randomized Adapative Search Procedure)

GRASP es una metaheurística que consiste, básicamente, en construir soluciones iniciales (método constructor aleatorio o semialeatorio) y aplicarles un método de mejora a lo largo de un determinado número de iteraciones o tiempo límite. La solución al problema será la mejor solución obtenida durante el proceso. Fue introducida por *Feo* y *Resende* en [12].

Ha sido utilizada ampliamente en el problema *MDP* [8, 2] donde las distintas variantes consisten en la aplicación de distintos tipos de métodos constructores y/o métodos de mejora. También se han realizado hibridaciones con otras técnicas como son *"Pathrelinking"* [41] o *"Minería de Datos"* [40].

Como se puede observar en el pseudocódigo 2.1, esta metaheurística es sencilla e intuitiva, pero se puede complicar todo lo que se quiera para intentar conseguir mejores soluciones. No olvidemos que la metaheurística es una guía para el diseño de procedimientos heurísticos.

```
1  Smejor = ConstruirSolucionAleatoria();
2  MIENTRAS (!CondicionDeParada()) {
3      Scandidata = ConstructorGreedyAleatorio();
4      Scandidata = BusquedaLocal(Scandidata);
5      SI (Valor(Scandidata) < Valor(Smejor))
6          Smejor = Scandidata;
7  }
8  DEVUELVE(Smejor);
```

Código 2.1: Algoritmo GRASP

2.3.2. SA (Simulated Annealing)

Este proceso, que se basa en el enfriamiento o templado de los metales, fue introducido por primera vez por Kirkpatric en [25], y aplicado al problema *MDP* por Kincaid en [24].

```
1  X = SolucionInicialFactible();
2  tmax = MaximoNumeroDeIteraciones;
3  q = TemperaturaInicialAlta;
4  MejorSolucion = X;
5  NumeroSoluciones = t = 0;
6
7  DObjetivo(X(t)) = CalcularValor(X(t));
8  MIENTRAS (HayMovimientoFactible(X(t)) Y (tmax < t)) {
9      CambioHecho = Falso;
10     MIENTRAS (NO CambioHecho) {
11         DX(t+1) = ElegirMovimientoFactible(X(t));
12         DObjetivo(X(t+1)) = CalcularValor(X(t),DX(t+1));
13         SI ((DObjetivo(X(t+1)) > DObjetivo(X(t))) O
14             (Potencia(e,(Dobjetivo(X(t+1))/q)) >= Random(0,1))) {
15                 X(t+1) = AplicarCambio(X(t),DX(t+1);
16                 CambioHecho = Verdad;
17         }
18     }
19     SI (DObjetivo(X(t+1) > DObjetivo(MejorSolucion))
20         MejorSolucion = X(t+1);
21     Si (Cambioq(t))
22         Decrementar(q);
23     t = t + 1;
24 }
25 Devolver(MejorSolucion);
```

Código 2.2: Algoritmo SA

Cómo se puede observar en el pseudocódigo 2.2, se intentan realizar movimientos (cambios en la solución) tal que el valor objetivo aumente o, en caso de no existir tal movimiento, aplicar aleatoriedad para salir de los mínimos locales que puedan alcanzarse. Tras un número de iteraciones determinado o cuando se llega a la no existencia de movimientos factibles se devuelve la mejor solución obtenida.

La función de la variable de temperatura (q) es la de posibilitar la ejecución de movimientos aunque no mejoren la función objetivo (línea 14). Inicialmente, tiene que tener un valor alto, por ejemplo un valor bastante mayor a la máxima distancia entre dos elementos del problema, para posibilitar la realización de cambios aunque no mejoren la solución. Con el paso del tiempo (o iteraciones), esta temperatura debe ir decrementándose para dificultar la realización de estos movimientos que no mejoran la solución. Esto es

así porque cuando el exponente al que se eleva el número e es próximo a 0 el valor resultante será próximo a 1, y cuando el exponente (que es siempre negativo porque el cambio no mejora la función objetivo) es menor que 0 el valor obtenido es menor de 1, más cercano a 0 cuanto mayor sea el valor absoluto del exponente, como se muestra en la siguiente formulación:

$$e^{(i)} \simeq 1 \;\; Si \;\; i \simeq 0 \tag{2.1}$$

$$0 < e^{(i)} < 1 \;\; Si \;\; i < 0 \tag{2.2}$$

Una opción bastante extendida para el cáclulo de q es calcularla en función del tiempo de ejecución del algoritmo según la siguiente función:

$$q(t) = k/\ln(1+t) \tag{2.3}$$

siendo k una constante calculada experimentalmente.

Si q disminuye lo suficientemente despacio, lo cual incrementa el tiempo de ejecución del algoritmo hasta tiempos inaceptables en determinados casos, se asegura la obtención de un óptimo global con una probabilidad $\simeq 1$ [21, 26].

2.3.3. TS (Tabu Search)

TS es una metaheurística muy utilizada en diversos campos desde hace mucho tiempo. Fue introducida por Glover en [16] y explicada en mayor profundidad en [17]. Su principal contribución es la de guiar la búsqueda heurística para explorar el espacio de soluciones más allá de los óptimos locales.

Básicamente, se basa en marcar como tabús ciertos movimientos o soluciones ya visitadas durante un cierto tiempo o capacidad de almacenaje de la lista tabú para impedir quedar atrapado en un óptimo local y ampliar el espacio de soluciones exploradas.

```
1  X = SolucionInicialFactible();
2  tmax = MaximoNumeroDeIteraciones;
3  MejorSolucion = X;
4  NumeroDeSoluciones = t = 0;
5  ListaTabu = ListaTabuVacia();
6
7  MIENTRAS (NO TodosMovimientosPosiblesSonTabu() O (t < tmax)) {
8      SI ((DX(t+1) = ElegirMejorMovimientoNoTabuFactible(X(t))) != Vacio)
9          X(t+1) = Modificar(X(t),DX(t+1));
10     SI (Valor(X(t+1)) > Valor(MejorSolucion))
11         MejorSolucion = X(t+1);
12     EliminarMovimientosTabuAntiguos(ListaTabu,t);
13     AgregarMovimientosTabu(X(t),X(t+1));
14     t = t + 1;
15 }
16 Devolver(MejorSolucion);
```

Código 2.3: Algoritmo TS

En el pseudocodigo mostrado en 2.3 se muestra el esquema básico de la metaheurística. Con este esquema se evitan ciclos a corto plazo (Short Term Memory) pero no se

asegura que se produzcan a largo plazo (Long Term Memory). Se han realizado múltiples adaptaciones de *TS* donde se intenta evitar los problemas a largo plazo de revisitar soluciones, así como otras variaciones o mejoras para la búsqueda de mejores soluciones, como pueden ser mejoras en los métodos de construcción, métodos de búsqueda local o reinicialización del proceso (Multistart), junto con hibridaciones con otros métodos.

Se puede encontrar más información sobre los distintos algoritmos basados en TS en [3] y aplicados a MDP en [30], así como un algoritmo *memético* (*memetic*) basado en TS (ejemplo de hibridación) en [42].

2.3.4. SS (Scatter Search)

SS es una metaheurística basada en población (population-based) cuyo algoritmo fue formulado definitivamente por Glover en [19] aunque, anteriormente, el mismo autor ya había esbozado la metaheurística en alguno de sus trabajos [15] y [18].

El objetivo de *SS* es mantener un conjunto de soluciones diversas y de alta calidad (*conjunto de referencia* o *Reference Set*) para poder ser recombinadas y refinadas con el ánimo de llegar a una solución óptima al problema tratado. Esta metaheurística está especialmente diseñada para problemas de optimización global de tipo combinatorio.

SS es de caracter iterativo, donde el conjunto de referencia evoluciona mediante la aplicación de cuatro métodos: "Generación de subconjuntos" (subset generation), "Actualización" (update), "Combinación" (combination) y "Mejora" (improvement).

```
1  ConjuntoInicial = ConstruirConjuntoInicial(); //Incluye mejora de las soluciones.
2  ConjuntoReferencia = ConstruirConjuntoReferencia(ConjuntoInicial); //Ordenadas.
3  NumeroIteraciones = t = 0;
4  NuevaSolucion = Verdad;
5
6  MIENTRAS (NuevaSolucion O (t < tmax)) {
7      NuevaSolucion = Falso;
8      SubConjuntos = GenerarSubConjuntos(ConjuntoReferencia);
9      Candidatos = Vacio;
10     PARA CADA SubConjunto EN SubConjuntos {
11         CandidatosRecombinados = RecombinaSubConjunto(SubConjunto);
12         PARA CADA CandidatoRecombinado EN CandidatosRecombinados {
13             Candidatos = Candidatos + Mejora(CandidatoRecombinado);
14         }
15     }
16     ConjuntoReferencia(t+1) = Selecciona(ConjuntoReferencia(t), Candidatos);
17     SI (ConjuntoReferencia(t+1) != ConjuntoReferencia(t))
18         NuevaSolucion = Verdad;
19     t = t + 1;
20 }
21 Devolver(ConjuntoReferencia);
```

Código 2.4: Algoritmo SS

El funcionamiento del algoritmo SS presentado en el pseudocódigo 2.4 es construir un conjunto de referencia a partir de una serie de soluciones obtenidas previamente. De entre este conjunto de soluciones obtenidas se escogen, en una proporción determinada, las mejores soluciones en cuanto a valor con respecto a la función objetivo y las más diversas entre sí (elementos distintos entre sí), siendo todas ellas distintas. El conjunto de referencia se ordena de mayor a menor valor de las soluciones incluidas (líneas 1 y 2).

Tras esto se inicia un proceso donde, iterativamente, se generan subconjuntos de soluciones basadas en algún criterio, se recombinan estos subconjuntos y se mejoran las soluciones recombinadas obtenidas para generar candidatos a entrar en el conjunto de referencia (líneas 8-14).

Posteriormente (línea 16), se evalúan los candidatos y se decide cuáles entran al conjunto de referencia (actualización) y qué soluciones tienen que salir de él.

El proceso termina cuando no hay nuevas soluciones añadidas al conjunto de referencia o cuando se alcance un número de iteraciones o tiempo predeterminado (línea 6).

SS es una metaheurística bastante flexible, en el sentido que permite una amplia libertad a la implementación de los distintos métodos que la conforman.

Se puede consultar en una mayor profundidad esta metaheurística en [31] y ver distintas variantes de la misma en [30]. En [13] se estudia el marco *SS* para analizar posibles métodos aplicables al mismo así como estructuras de memoria, y se crea una solución al problema *MDP*, bastante efectiva según los autores.

2.3.5. PR (Path Relinking)

PR, más bien que una metaheurística en sí, es una forma generalizada de *SS* [19] donde se persigue el mismo objetivo: conseguir más información de la que por sí solo tiene un conjunto de soluciones para llegar a un objetivo mejor. Al igual que *SS* es un método evolutivo o basado en población. A menudo se la considera como un método *SS* donde la estrategia de combinación se realiza mediante *PR*. Por lo tanto, se comparte el esquema y es totalmente aplicable el pseudocódigo en el listado 2.4.

La idea de *PR* es recorrer el camino (path) entre dos soluciones buenas para explorar los distintos pasos del camino y poder obtener otras soluciones mejores a las que sirven como extremos del camino. Con esto se consigue explorar parte del *vecindario* (*neighborhood*) y mediante el proceso de mejora poder expandirse a otros vencindarios.

Muy a menudo, tanto *SS* como *PR* son combinados con metodologías *TS* para favorecer la diversidad y no incidir en soluciones ya visitadas, frente a la intensificación (inclusión en el conjunto de referencia de las mejores soluciones y varias etapas de mejora aplicadas) evidente en los métodos *SS* y *PR*.

El funcionamiento básico de *PR* es escoger dos buenas soluciones y establecer una de ellas como "solución de partida" y otra como "solución guía". Sucesivamente, se van sustituyendo atributos de la "solución de partida" por los de la "solución guía", según un criterio escogido, lo que da como resultado distintas soluciones de la trayectoria entre ambas. Estas soluciones son susceptibles de ser mejoradas mediante un método de mejora. El objetivo final es encontrar entre estas soluciones de la trayectoria algunas que mejoren a las que originaron la trayectoria.

Más información sobre el planteamiento tanto de *SS* como *PR* así como estrategias y factores a tener en cuenta se pueden encontrar en [19]. En [41] se propone un algoritmo *GRASP* con *PR* aplicado a *MDP*.

2.3.6. VNS (Variable Neighborhood Search)

VNS es una metaheurística de, relativamente, reciente introducción. Fue introducida por Hansen-Mladenović en 1997 [36]. La idea básica que guía *VNS* es el cambio sistemático de *entornos* o "*vecindarios*" en una búsqueda local.

```
1  SeleccionarConjuntoEntornosAgitacion Nk para k = 1..kmax;
2  SeleccionarConjuntoEntornosDescenso Nj para j = 1..jmax;
3  X = GenerarSolucionInicial();
4
5  MIENTRAS (NO CondicionDeParada()) {
6      k = 1;
7      MIENTRAS (k < (kmax + 1)) {
8          X1 = Agitar(Nk(X)); //Shake.
9          //Aplicar busqueda local VND (VNS ascendente).
10         j = 1;
11         MIENTRAS (j < (jmax + 1)) {
12             X2 = MejorSolucion(Nj(X1));
13             SI (X2 > X1) {
14                 X3 = X2;
15                 j = 1;
16             } SI NO
17                 j = j + 1;
18         }
19         SI (X3 > X) {
20             X = X3;
21             k = 1;
22         } SI NO
23             k = k + 1;
24     }
25 }
26 Devolver(X);
```

Código 2.5: Algoritmo VNS General

En el pseudocódigo en 2.5 se muestra una de las variantes del algoritmo *VNS* llamada *VNSGeneral*. Existen otras variantes (*VNS Descendente*, *VNS Básico*, *VNS Reducida*...) del algoritmo pero la estructura es básicamente la misma. En esta variante se puede observar como se aplica anidada dentro del propio algoritmo VNS la variante VNS ascendente.

Un *vecindario* de x, siendo x una solución del espacio de soluciones X, consiste en todas las soluciones del problema vecinas de x en función de alguna métrica o métricas definida sobre X. Por ejemplo, podríamos considerar un vecindario de x donde todas las soluciones que lo integren varían en un sólo elemento con respecto a la solución x. A este vecindario también se le llama estructura de entorno, y varios vecindarios distintos suponen distintas estructuras de entorno.

La metaheurística *VNS* está basada en tres hechos simples:

1. Un óptimo local con una estructura de entornos no lo es necesariamente con otra.
2. Un óptimo global es un óptimo local con todas las posibles estructuras de entornos.
3. Para muchos problemas, los óptimos locales con la misma o distinta estructura de entornos están relativamente cerca.

La última información ha sido comprobada de forma empírica. Se sabe que los óptimos locales proporcionan información sobre el óptimo global aunque no se sepa exac-

tamente cual es esa información. Por lo tanto, se supone que es conveniente explorar el espacio de soluciones próximo a los máximos locales. De hecho, esta es la principal característica de los distintos algoritmos *VNS*.

Se han propuesto muchas variantes de *VNS*, principalmente basadas en variaciones del método de "agitación" (*Shake*) y del método de búsqueda local. En [3] se pueden consultar ejemplos de aplicación y variantes de *VNS* y en [30] concretamente aplicaciones a *MDP*, así como una especificación más amplia de la heurística y sus distintas variantes e hibridaciones en [22].

2.3.7. Otras metaheurísticas e hibridaciones

Puede consultarse en [3] un repaso y ampliación de otro tipo de metaheurísticas de optimización combinatoria e hibridaciones con otros métodos de optimización. En este estudio se tratan y muestran aplicaciones concretas de metaheurísticas de optimización combinatoria basadas en población (population-based) como los *Algoritmos Genéticos* (*Genetic Algorithms*), la *Optimización de Colonia de Hormigas* (*Ant Colony Optimization*) o la *Optimización de Partículas en Enjambre* (*Particle Swarm Optimization*).

También se tratan hibridaciones de metaheurísticas como los algoritmos *Meméticos* (hibridación entre algoritmos genéticos y *Búsqueda Local*. *Memetic Algorithms*) e hibridaciones con otras técnicas como *Programación de Restricciones* (*Hybridizing Metaheuristic With Constraint Programming*) o *Búsqueda en Árboles* (*Hybridizing Metaheuristic With Tree Search Technique*).

2.4. Conclusión

En este capítulo se han analizado las características principales de distintas metaheurísticas, y se incluyen referencias a trabajos donde estos métodos se han aplicado al problema que nos ocupa. No se trata de realizar un recorrido exhaustivo por todas las posibles heurísticas y metaheurísticas que se podrían aplicar al problema, sino de analizar la forma de trabajo de los distintos métodos para utilizar algunas de estas ideas en el desarrollo del algoritmo parametrizado que desarrollamos en el siguiente capítulo para el problema *MDP*.

Capítulo 3

Aplicación de heurísticas al MDP

En este capítulo vamos a presentar un nuevo algoritmo realizado para la resolución del problema *MDP*.

El algoritmo, al que llamaremos **GRASP_M**, es una variante de la metaheurística *GRASP* con aportaciones encaminadas a conseguir rápidamente buenos resultados para el problema *MDP*.

En un primer apartado se analizará la estructura del algoritmo **GRASP_M**, su filosofía y las estructuras de datos utilizadas para llevarlo a cabo.

En un segundo apartado se realizará un estudio experimental sobre los parámetros de funcionamiento del algoritmo y la mejor elección de sus valores por defecto, y se analizará el algoritmo en términos de "*fitness*" y *tiempos de ejecución* para comprobar su calidad.

3.1. El algoritmo GRASP_M

El algoritmo **GRASP_M** está basado en la metaheurística *GRASP* y, por tanto, consiste en ir generando con un método constructor soluciones iniciales que pueden o no ser mejoradas en función de distintos criterios. Si la solución inicial es seleccionada para aplicársele el proceso de mejora se le aplica y este proceso puede incluir cambios en todos los m elementos de la solución, es decir, no se aplica un límite de cambios a los elementos en la solución inicial. El algoritmo se ha especificado como multi-arranque en función de un número de mejoras aplicadas sin cambio en la mejor solución encontrada hasta el momento desde el último arranque del algoritmo. Se ha observado, a través de experimentación, que no es aceptable, ni en términos de costo computacional ni en ganancia de la solución, el seguir el proceso de búsqueda de una solución mejor y es mejor reiniciar el algoritmo desde 0, almacenando la solución en un conjunto de soluciones si procede. El pseudocódigo del algoritmo se muestra en el listado de pseudocódigo 3.1.

El algoritmo comienza con la inicialización de las variables (líneas 1 a 3 y 5). Estas variables son:

- **SolucionActual**. Esta variable contendrá la mejor solución generada por el proceso *GenerarNuevaSolucion()* durante una determinada ejecución del algoritmo (desde

```
SolucionActual = SolucionMejora = Vacia;
SolucionPropuesta = SolucionMejoraTemp = Vacia;
ConjuntoSoluciones = Vacio;
PrepararProblema(P);
TiempoAlgoritmo = 0;
MIENTRAS (TiempoMaximoAlgoritmo > TiempoAlgoritmo) {
    SolucionPropuesta = GenerarNuevaSolucion(P,SolucionPropuesta);
    SI ((SolucionPropuesta > SolucionActual) || CriterioMejora()) {
        SI (SolucionPropuesta > SolucionActual)
            SolucionActual = SolucionPropuesta;
        SolucionMejoraTemp = MejorarSolucion(SolucionPropuesta);
        SI(SolucionMejoraTemp > SolucionMejora) {
            SolucionMejora = SolucionMejoraTemp;
            iteracionesSinMejora = 0;
        SI NO
            IteracionesSinMejora = IteracionesSinMejora + 1;
        SI (CriterioCambioSolucionPropuesta())
            SolucionPropuesta = SolucionMejora;
        SI NO SI (CriterioReinicio()) {
            IncluirSolucionEnConjunto(ConjuntoSoluciones,SolucionMejora);
            SI (CriterioReinicioConElementos())
                SolucionPropuesta = SeleccionarElementos(ConjuntoSoluciones);
            SI NO
                SolucionPropuesta = Vacia;
            ReiniciaAlgoritmo();
        }
    }
    TiempoAlgoritmo = CalcularTiempoTranscurrido();
}

IncluirSolucionEnConjunto(ConjuntoSoluciones,SolucionMejora);
P->ConjuntoSoluciones = ConjuntoSoluciones;
Devolver(P);
```

Código 3.1: Algoritmo GRASP_M

último arranque/rearranque del algoritmo). Inicialmente no contiene ninguna solución.

- **SolucionMejora**. Albergará la mejor solución, tras aplicársele el proceso *MejorarSolucion()*, hasta el momento dentro de una determinada ejecución del algoritmo. Inicialmente no contiene ninguna solución.

- **SolucionPropuesta**. Contendrá la solución actual generada por el proceso *GenerarNuevaSolucion()*, una solución de partida para una determinada ejecución del algoritmo o una solución mejorada nueva obtenida durante la ejecución del algoritmo. Su valor servirá de base para la generación de nuevas soluciones. Inicialmente no contiene ninguna solución.

- **SolucionMejoraTemp**. Contendrá la solución obtenida tras aplicar a *SolucionPropuesta* el proceso *MejorarSolucion()*. Es una variable de utilidad temporal para realizar comparaciones y en su caso asignaciones. Inicialmente no contiene ninguna solución.

- **ConjuntoSoluciones**. Esta variable servirá para almacenar las mejores soluciones encontradas durante las distintas ejecuciones (arranques) del algoritmo. Tiene varias utilidades en el algoritmo (véase 3.1.1.3). Inicialmente está vacía.

- **TiempoAlgoritmo**. Esta variable servirá para contar el tiempo desde el inicio del algoritmo (reloj del sistema) y así poder saber el tiempo en que se guardan las soluciones en el conjunto de soluciones (con cada rearranque del algoritmo) y cuando se debe terminar la ejecución del algoritmo aplicado a un problema determinado. Se calcula una vez por iteración del algoritmo y su valor está expresado en milisegundos.

También, antes de la ejecución del algoritmo, se realiza una fase de preparación inicial (línea 4) del problema *MDP*. En ella se configuran todos los aspectos parametrizables del problema según se hayan establecido antes de la ejecución del algoritmo y se crean y preparan las estructuras de datos (dinámicas) necesarias para su ejecución. Se verá en más detalle en 3.1.1. También se establece la semilla aleatoria para la función "**rand()**" de C que no se cambiará durante toda la ejecución del algoritmo.

En el listado de pseudocódigo 3.1 se puede observar como se van generando soluciones (**GenerarNuevaSolucion()**) en cada iteración del algoritmo (se verá más adelante en 3.1.2 el modo como se hace) y si hay mejora de la solución generada en la iteración actual con respecto a la mejor generada hasta el momento (desde el último arranque del algoritmo) se guarda con el propósito de comprobar la posterior mejora de soluciones generadas (líneas 7 a 10), que tiene su importancia en el criterio para aplicar el proceso de mejora a las soluciones generadas en cada iteración como veremos a continuación.

La mejora de soluciones no se realiza para todas las soluciones generadas sino que o bien tiene que haber una mejora con respecto a la mejor solución generada hasta el momento o tiene que cumplirse un criterio para forzarla (línea 8). El criterio puede ser múltiple y su función principal es la de evitar un gran número de iteraciones sin mejora pues no se estaría explorando eficientemente el espacio de soluciones del problema. A su vez, como la mejora de soluciones (3.1.3) es la parte del algoritmo más costosa computacionalmente tampoco conviene realizar un excesivo número de mejoras.

El criterio para forzar la mejora de una solución producida por el generador de soluciones, que no mejore la mejor solución generada hasta el momento, podría ser el mejorar las soluciones iniciales en las que haya elementos del problema que han entrado por primera vez en una solución generada (nunca antes habían sido incluidos en una solución generada en el actual arranque del algoritmo). También se puede establecer el criterio de forzar la mejora de soluciones generadas estableciendo un porcentaje mínimo de aplicaciones de mejora con respecto al número total de soluciones generadas. Estos criterios son utilizados en este algoritmo, junto con otros, en función de una serie de parámetros.

Si la solución mejorada, obtenida de la aplicación del proceso de mejora (**MejorarSolucion()**) a la solución generada actual, supera en calidad, con respecto a la función objetivo, a la mejor solución mejorada hasta el momento, se almacena como mejor solución mejorada y se restablece el contador que registra el número de veces consecutivas que una solución generada tras el proceso de mejora no ha superado a la mejor solución mejorada almacenada, en caso contrario se incrementa dicho contador (líneas 12 a 16). Veremos, un poco más adelante, que este contador tiene importancia a la hora de decidir si reiniciar el algoritmo o no.

Son interesantes las líneas 17 y 18 del algoritmo, donde la solución generada actual (*SolucionPropuesta*) puede ser cambiada a la mejor solución mejorada encontrada hasta el

momento para que, a partir de ella, se generen nuevas soluciones, es decir, partiríamos de una mejor solución desde donde crear nuevas soluciones generadas con el generador de soluciones. Este cambio no es automático pues se ha comprobado experimentalmente que no es una buena idea hacerlo con cada mejor solución mejorada encontrada y tampoco no hacerlo nunca. Por eso se decide en base a un criterio que, en este caso, es aleatorio. El proceso de cambio de la solución generada actual es el siguiente: si tras una mejora de la solución generada, esta es la mejor solución mejorada hasta el momento, se decide aleatoriamente (1 o 0) si cambiar o no la actual solución generada a la mejora encontrada. Esto podría causar que quedáramos atrapados en algún tipo de óptimo local pero, gracias al multireinicio del algoritmo y a que se pueden introducir un gran número de cambios a las soluciones generadas a través del generador de soluciones, no es un hecho preocupante para la búsqueda de soluciones buenas al problema *MDP*.

En la línea 19 del algoritmo se comprueba el criterio para reiniciar el algoritmo en búsqueda de una solución nueva al problema *MDP*. Básicamente, consiste en comprobar si ha habido un determinado número de procesos de mejora consecutivos que no han conseguido mejorar a la mejor solución mejorada encontrada desde el último inicio/reinicio del algoritmo. Aquí interviene el valor **IteracionesSinMejora** calculado previamente. En los siguientes pasos (líneas 21 a 26) se decide entre dos métodos de reinicio previo almacenamiento de la mejor solución actual en el conjunto de soluciones si procede. Hay que decir que se ha decidido no aplicar el reinicio del algoritmo para problemas considerados de tamaño pequeño (con un valor de $n \leq 400$ o $m \leq 40$) porque, generalmente, se consigue una muy buena solución con una sóla ejecución (arranque) del algoritmo. Desde luego, se ha comprobado con el conjunto de datos de referencia [33] que se tiene pero podría ser que con otros conjuntos de datos esto no fuera así.

Los dos tipos de reinicio del algoritmo consisten o bien en comenzar completamente desde cero sin tener en cuenta nada de la ejecución/es anterior/es o bien escoger los elementos coincidentes (línea 22) de un determinado número de las mejores soluciones del conjunto de soluciones almacenadas hasta el momento. Obviamente, para esto último, deben haber, por lo menos, 2 soluciones en el conjunto de soluciones (**ConjuntoSoluciones**). Tras cierta experimentación, se ha decidido reiniciar mediante este método en el primer reinicio del algoritmo en que ya tengamos dos soluciones en el conjunto y, a partir de ahí, cada 4 reinicios completos se realiza un reinicio con elementos coincidentes. Esto es así porque se ha comprobado que es un proceso costoso en tiempo de procesamiento y pocas veces, aunque sí en determinados casos, se obtienen mejores soluciones que las ya incluidas en el conjunto de soluciones hasta el momento. Por esto se hace al principio y cada 4 iteraciones para dar la oportunidad de que varíen las mejores soluciones del conjunto de soluciones, pues con el reinicio completo se suelen encontrar soluciones bastante buenas. Si el número de elementos coincidentes entre las soluciones del conjunto de soluciones seleccionadas supera el valor de $0{,}80 * m$, se incluye una solución más del conjunto de soluciones (si la hay) para seleccionar los elementos coincidentes entre ellas, en caso contrario se excluye una solución siempre y cuando se tenga un mínimo de dos soluciones para calcular los elementos coincidentes. Con este proceso se intenta realizar una intensificación en la búsqueda de un máximo global para el problema *MDP* basándonos en el principio de que los máximos locales y el global deben compartir cierta información y estar relativamente cerca en cuanto a estructura de vecindarios, uno de los principios en los que se basa la metaheurística *VNS* (2.3.6).

Para una aplicación efectiva del método de reinicio de soluciones con elementos, se debe disponer de un tiempo suficiente para que se produzcan varios reinicios del algoritmo. Este tiempo depende del tamaño del problema que si es grande, evidentemente, necesitará de tiempos más largos. Si se quiere desactivar este sistema basta con configurar el tamaño del conjunto de soluciones a 1.

Las soluciones obtenidas en las distintas ejecuciones del algoritmo se incluyen en un conjunto de soluciones (líneas 20 y 31) que tiene un tamaño predefinido (por defecto 5) al comenzar el algoritmo *MDP*. Este conjunto de soluciones se mantiene tanto para posibilitar el reinicio del algoritmo con elementos coincidentes como para ser devuelto como parte de la información obtenida por la ejecución del algoritmo. El criterio para que una solución sea introducida en el conjunto es que no sea igual (mismo valor de la solución y mismos elementos del problema pertenecientes a la solución) a otra solución ya incluida en el conjunto, y que si el conjunto se encuentra lleno no sea menor a todas las soluciones del mismo. Las soluciones del conjunto se encuentran ordenadas en orden descendente, es decir, de mayor a menor valor de la solución almacenada.

El hecho de que se intente incluir la mejor solución actual tras terminar el algoritmo (línea 31) es así porque tras cumplirse el tiempo límite del algoritmo, muy probablemente, no ha sido almacenada al no haberse producido un reinicio del mismo (momento en que se almacenan las soluciones obtenidas por el algoritmo). Con los criterios de inclusión, anteriormente expuestos, esta solución será o no será incluida en el conjunto. No sería descabellado pensar que esta última solución fuera lo suficientemente buena como para entrar a formar parte del conjunto de soluciones.

Como última acción del algoritmo, tras haber liberado toda la memoria dinámica asignada, se devuelve toda la información relevante de la ejecución del algoritmo (línea 33). La información devuelta comprende el conjunto de mejores soluciones obtenidas, la configuración del problema (datos de configuración para la ejecución del algoritmo) e información sobre los tiempos de ejecución del algoritmo (véase 3.2).

3.1.1. La fase de preparación

Como ya se ha comentado antes, en esta fase del algoritmo se aplica la configuración así como la preparación de todas las estructuras de datos necesarias para su correcta ejecución.

3.1.1.1. Configuración E/S

En la configuración del algoritmo intervienen tanto parámetros para una determinada forma de ejecución del algoritmo como para la obtención de la entrada/salida del mismo en término de ficheros.

En términos de entrada/salida los parámetros serían el fichero de distancias desde donde leer las distancias asociadas al problema en el formato que se provee en [33], el fichero de salida donde se mostrarán los datos referentes a la resolución del problema y el modo en que se añadirán estos datos al fichero de salida: “sobreescribir” o “añadir”.

Se puede obviar el crear/añadir un fichero de salida para exportar los resultados del algoritmo especificando una cadena de longitud cero para el fichero de salida (”\0”) o

simplemente no especificándolo como parámetro. Si no se especifica un fichero de salida la información es mostrada por la salida estándar.

En cuanto al formato de salida de los resultados del algoritmo, este será en un fichero de valores separados por comas (".csv"). Los campos o valores separados por comas serán de la siguiente forma (entre paréntesis se especifica el orden de los distintos campos en la lista separada por comas):

(1) Nombre del fichero de entrada de distancias. Es un texto descriptivo que identifica el problema tratado.

(2) Número de elementos en el problema tratado (n). Tamaño del problema.

(3) Número de elementos pertenecientes a la solución (m). Tamaño de la solución.

(4) Tiempo máximo del algoritmo especificado por configuración en segundos.

(5) Tiempo empleado por el proceso *MDP* (incluyendo fase de preparación, finalización del proceso *MDP* y salida de resultados) en milisegundos.

(6) Tiempo empleado por el algoritmo (sin incluir fase de preparación, finalización del proceso y retorno de resultados) en milisegundos.

(7) Tiempo empleado por el algoritmo en encontrar la mejor solución en milisegundos.

(8) Valor de la mejor solución encontrada por el algoritmo.

(9..(m+9)) Elementos ordenados en sentido ascendente pertenecientes a la mejor solución separados por comas.

Se ha escogido este formato de salida para que los datos aportados por el algoritmo sean fácilmente exportables y utilizables.

3.1.1.2. Configuración del algoritmo

En cuanto a los valores de parametrización del algoritmo se pasan directamente a través de la definición del problema (Ver código en 3.2).

Los valores n y m se asignan tras la lectura del fichero de distancias del problema, donde n es el tamaño del problema y m el tamaño de la solución.

El valor de MDP_NSC es el tamaño del conjunto de soluciones y es configurable por el usurario. Si su valor es menor de 1 se le asigna el valor por defecto consignado en la aplicación que es de 5 (modificable mediante constante definida en el fichero "datosMPD.h").

Se ha incluido, tras la ejecución del algoritmo, la posibilidad de realizar un proceso de *PR* (*Path Relinking*. Véase 2.3.5) sobre el conjunto de soluciones obtenidas tras la ejecución del algoritmo. Si mediante MDP_FLAGS (ver más adelante) se decide su ejecución, el mecanismo es aplicar PR a cada par de soluciones distintas dentro del conjunto de soluciones. Con la información obtenida tras la experimentación se ha observado que el valor de mejora, si la hay, es muy poco con respecto a la mejor solución obtenida. Por

```
struct MDP_Problema_Estructura {

    int n, m;
    int MDP_NSC; //Numero de soluciones maximas a almacenar.
    elementoConjuntoSoluciones *MDP_conjuntoSoluciones; //Soluciones almacenadas.
                                                //Creado por la funcion en base a  MDP_NSC.
    elementoConjuntoSoluciones MDP_PR; //Solucion PR final obtenida si procede.
    unsigned int MDP_flags; //Flags de configuracion del tratamiento del problema.
    double MDP_PM; //Porcentaje de soluciones a mejorar. Porcentaje de 0..1 (tanto por
        uno).
    int MDP_MISM; //Maximo únmero de iteraciones sin mejora antes de reiniciar ósolucin.
    int MDP_PREC; //Precision en la comparacion de soluciones (cifras decimales) 0..10.
    int MDP_TMAX; //Tiempo maximo para aplicar algoritmo MDP en segundos.
    long long MDP_TEP; //Tiempo ejecucion total proceso MDP.
    long long MDP_TEA; //Tiempo ejecucion algoritmo MDP.
    int retorno; //Resultado de ejecucion de la funcion.
};

typedef MDP_Problema_Estructura MDP_Problema;
```

Código 3.2: Estructura definitoria problema MDP

eso, por defecto, esta opción se mantiene desactivada pero si lo estuviera se devolvería en *MDP_PR* la mejor solución obtenida mediante la aplicación de *PR* al conjunto de soluciones obtenidas tras el algoritmo. Debe mencionarse que el algoritmo implementa dos tipos distintos de *PR*. Uno, el que realiza simplemente el recorrido de la trayectoria entre las dos soluciones (soluciones guía y partida) y, dos, el que realiza este recorrido pero aplicando mejoras a cada solución intermedia obtenida. El primero es menos costoso (más rápido) en cuanto a términos de procesamiento y el segundo es bastante más costoso debido al proceso de mejora realizado. Se deja a experimentación de quien esté interesado el uso de un método u otro. Por defecto, se elige la primera opción. El cambio del método requiere del cambio del código fuente del algoritmo.

El valor *MDP_PM* es un valor en el intervalo $(0, 1)$ que establece el porcentaje mínimo de mejoras a realizar a las soluciones generadas tal como se explicaba en 3.1. Su funcionamiento es simple: si el número de aplicaciones de mejoras con respecto al número de soluciones generadas cae por debajo de este valor se fuerza a que se produzcan mejoras de la solución generada, independientemente de si la solución generada es o no mayor que la mejor solución generada hasta el momento de la actual ejecución del algoritmo (último arranque del algoritmo). Un valor de 0,05, es decir, un 5 % de mejoras sobre las soluciones generadas parece funcionar bien.

MDP_MISM indica el máximo número de iteraciones en las que se aplica el proceso de mejora a las soluciones generadas pero no se mejora la mejor solución encontrada hasta el momento en la actual ejecución del algoritmo. Si se alcanza dicho límite (50 por defecto) se aplicará un reinicio del algoritmo (multiarranque).

MDP_PREC indica la precisión en la comparación de valores, sobre todo en la de los valores de las soluciones. Se ha observado que el tipo de dato *"double"*del lenguaje *C* no es muy preciso en cuanto a comparación de distintos valores que han sufrido numerosos cálculos como sucede en este algoritmo. Se puede indicar una precisión de comparación entre 0 y 10 caracteres decimales. Una precisión de 0 indica que los valores deben diferenciarse en más de una unidad para ser distintos, y un valor *k* entre 1 y 10 indica que el

valor de la k-ésima cifra decimal tiene que ser igual para considerar los valores iguales, es decir, la diferencia de las dos cantidades comparadas debe ser menor a 10^{-k}.

El valor *MDP_TMAX* especifica la duración máxima del algoritmo en segundos. Los valores *MDP_TEP* y *MDP_TEA* son asignados tras la ejecución del algoritmo *MDP* y contienen, respectivamente, los tiempos empleados por el proceso completo y el algoritmo en sí en milisegundos.

El valor *retorno* indica el estado tras la ejecución del algoritmo. Si este valor es distinto de 0 es que ha habido algún problema en la ejecución del algoritmo y no son fiables los valores devueltos por el mismo. No se han especificado aún mensajes informativos para identificar el tipo del problema.

Mención aparte tiene el campo *MDP_FLAGS*. Como su propio nombre indica está compuesto de unas banderas (flags) que afectan a la ejecución del algoritmo. Estas banderas son las especificadas en el listado de código 3.3, y se comentan brevemente a continuación:

```
1  //MDP_Flags
2
3  #define MDP_FSDO 1 //Orden en vector Suma de Distancias.
4  #define MDP_FSDOD 2 //Orden Descendente en vector Suma de Distancias.
5  #define MDP_FSDOA 4 //Orden Ascendente en vector Suma de Distancias.
6  #define MDP_FSDOR 8 //Orden Aleatorio (Random) en vector Suma de Distancias.
7  #define MDP_FDOO 16 //Orden en matriz ordenada.
8  #define MDP_FDOOD 32 //Orden Descendente en matriz ordenada.
9  #define MDP_FDOOA 64 //Orden Ascendente en matriz ordenada.
10 #define MDP_FEPS 128 //Aplicar Mejora a elementos por primera vez en solucion inicial.
11 #define MDP_FNPM 256 //Aplicar "nodo prohibido" en mejora de soluciones iniciales.
12 #define MDP_FPR 512 //Aplicar PR a conjunto de soluciones finales.
13 #define MDP_FPRM 1024 //Aplicar mejora a soluciones intermedias PR.
14 #define MDP_FRSAMR 2048 //Reiniciar solucion inicial a mejora aleatoria cuando se mejore
       mejor solucion.
15 #define MDP_FNOVALIDO 4096 //Flags no validos.
```

Código 3.3: Flags para el problema MDP

- *MDP_FSDO* indica si el vector de suma de distancias (se verá más adelante en la explicación de las estructuras de datos en 3.1.1.3) debe ordenarse o no. *MDP_FSDOD*, *MDP_FSDOA* y *MDP_FSDOR* indican si este vector debe ordenarse de forma descendente, ascendente o aleatoria, respectivamente.

- *MDP_FDOO* indica si la matriz de distancias ordenada (se verá más adelante en la explicación de las estructuras de datos en 3.1.1.3) debe ordenarse o no. *MDP_FDOOD* y *MDP_FDOOA* indican si esta matriz debe ser ordenada de forma descendente o ascendente, respectivamente.

- *MDP_FEPS* indica al algoritmo si forzar la mejora de las soluciones generadas, que aunque no mejoren la mejor solución generada hasta el momento, incluyen elementos del problema por primera vez, en esta ejecución del algoritmo, en una solución generada. Esto forma parte del criterio de aplicación de mejoras explicado en 3.1. Es totalmente conveniente hacerlo así para evitar saltarse posibles cambios en las soluciones generadas que sean susceptibles de mejorar en el proceso de mejora, aunque las mayores mejoras se suelen obtener con un mayor número de cambios

en la solución generada. De cualquier forma, si se procede de esta manera se asegura que ningún elemento que entre en una solución generada por primera vez sea obviado en el proceso de mejora. La experimentación realizada en 3.2.1 indicará si es conveniente o no aplicar este criterio.

- *MDP_FNPM* indica que el último elemento en entrar a la solución generada actual no debe ser extraído de ninguna manera de la solución mejorada obtenida durante el proceso de mejora. Se consideró, inicialmente, como una opción factible durante la realización del algoritmo, el obligar a no salir de la solución mejorada el último elemento introducido en la solución generada a mejorar, pero tras diversas experimentaciones de la ejecución del algoritmo se observaron mejores resultados desactivando esta opción. De cualquier manera, se mantiene esta opción para futuras experimentaciones con el algoritmo. Por defecto, esta opción está desactivada.

- *MDP_FPR* indica si se realizará o no la aplicación de la metaheurística *PR* al conjunto de soluciones finales.

- *MDP_FPRM* indica que se aplicará la metaheurística *PR* cada vez que se produzca una solucion en el proceso de mejora que mejore la mejor solución mejorada encontrada hasta el momento. No se ha experimentado mucho con esta opción pero no se considera eficiente en cuanto a la obtención de mejores soluciones obtenidas. De cualquier forma queda sujeto a mayor experimentación.

- *MDP_FRSAMR* indica si se debe reiniciar la actual solución generada de forma aleatoria a la mejor solución mejorada obtenida en la actual iteración del algoritmo o, en caso contrario, reiniciar siempre la actual solución generada, cada vez que se produzca una mejora de la mejor solución obtenida hasta el momento, a la mejor solución obtenida en la actual iteración (último arranque) del algoritmo. Por defecto se opta por la opción de asignación fija.

- A la bandera *MDP_FNOVALIDO* todavía no se le ha asignado una utilidad específica.

Cualquier valor de bandera no especificado explícitamente conservará sus valores por defecto.

Se han definido algunas macros útiles para el tratamiento (consulta, modificación, ...) de las banderas (Consultar el fichero "datosMDP.h").

3.1.1.3. Estructuras de datos

Las estructuras de datos utilizadas para la resolución del problema *MDP* se han creado siempre teniendo en cuenta los criterios de eficiencia, sobre todo de espacio de almacenamiento, y utilidad evidente para el fin al que están destinadas.

La primera de las estructuras de almacenamiento utilizadas es la que almacena la **matriz de distancias** perteneciente al problema *MDP* concreto tratado. Estas matrices formadas por las distancias de los elementos del problema entre sí ($d(i,j)$) tienen una peculiariedad y es que:

$$d(i,j) = d(j,i) \quad \forall i,j \in N \tag{3.1}$$

$$d(i,i) = 0 \tag{3.2}$$

Esto significa que la distancias entre dos elementos i y j es la misma independientemente del orden de los elementos entre sí (ecuación 3.1) y que la distancia de un elemento i consigo mismo es 0 (ecuación 3.2).

Estas dos peculiaridades provocan que la matriz de distancias sea una matriz triangular (superior o inferior según gustos), como la mostrada en la figura 3.1, en la que la diagonal está formada por ceros pues correspondería a la distancias de cada elemento consigo mismo.

Por tanto, si queremos reservar espacio para una matriz triangular deberíamos reservarlo para un número de elementos $n*(n+1)/2 = \sum_{i=1}^{n} i$, incluida la diagonal que ya veremos que se utiliza para otros propósitos puesto que sus valores en cuanto a distancias son siempre 0.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & . & . & . & a_{1n} \\ 0 & a_{22} & a_{23} & . & . & . & a_{2n} \\ 0 & 0 & a_{33} & a_{34} & . & . & a_{3n} \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ 0 & 0 & 0 & 0 & 0 & 0 & a_{nn} \end{bmatrix}$$

Figura 3.1: Matriz triangular superior

De esta forma tenemos que acceder a las distancias almacenadas, tanto para su modificación como su consulta, en un vector lineal de la siguiente forma para una distancia $d(i,j)$ siendo $i \leq j$:

$$d(i,j) \;=\; vector\,[i*n-(i*(i+1)/2)+j] \;\; \forall\, 0 \leq i \leq j \leq n-1 \tag{3.3}$$

Esta expresión proviene de la simplificación de la siguiente expresión:

$$d(i,j) \;=\; vector\,[i*(n+1)-(i*(i+1)/2)+(j-i)] \;\; \forall\, 0 \leq i \leq j \leq n-1 \tag{3.4}$$

que es equivalente a :

$$d(i,j) \;=\; vector\,[i*n+i-(i*(i+1)/2)+(j-i)] \;\; \forall\, 0 \leq i \leq j \leq n-1 \tag{3.5}$$

donde $+i$ se elimina con la i de $(j-i)$ y se convierte en la fórmula 3.3.

El origen de la fórmula 3.4 se obtiene de restar a la posición $i * (n + 1)$, que ocuparía el elemento en la diagonal (primer elemento de matriz de distancias) de la matriz para la fila i, el número de elementos, hasta la fila i, $i * (i + 1)/2$ que no se utilizan por ser la matriz triangular. De esta forma se transforma la posición que un elemento ocuparía en la matriz a una posición en un vector lineal de elementos sin pérdida de espacio, es decir, el vector lineal de elementos tan sólo contiene un número de elementos $n * (n + 1)/2$, que es el número de elementos de la matriz triangular distintos de 0 (excepto la diagonal que es 0 pero se utiliza para almacenar la suma de distancias para la fila i). j indica la posición de un elemento en la fila i de la matriz triangular siendo $j \geq i$.

Como habíamos dicho, los valores de la diagonal de la matriz de distancias siempre son cero y, por tanto, no sirven para nada en la resolución del problema *MDP*. Se utiliza el valor de estos elementos para almacenar la suma de distancias de un elemento i correspondiente a una fila con respecto a todos los demás elementos j del problema, es decir, un número de elementos igual a $n - 1$. El cálculo de estas sumas se realiza en el proceso de lectura del fichero de distancias. Veremos la utilidad que se da a estos valores más adelante.

Considero que la organización de la matriz de distancias realizada de esta manera es una forma eficiente de almacenamiento de las distancias y con un acceso fácil. En principio, el almacenamiento de las distancias debería hacerse en una zona consecutiva de la memoria si no hay problemas en la capacidad de almacenamiento.

Para hacer lo más versátil posible la estructura de almacenamiento de distancias se ha optado por el tipo de datos "double", capaz de albergar una gran diversidad de valores.

Para simplificar el acceso a la matriz de distancias se ha creado una macro que obtiene las distancias tomando como parámetros el *vector de la matriz de distancias*, n, i y j.

Otra estructura de datos creada y necesaria para el proceso de generación de soluciones del algoritmo es la de una **matriz de distancias ordenadas**. Esta matriz también es un vector organizado de la misma forma que el *vector de distancias*, explicado anteriormente, pero difiere en el tipo de elementos que alberga. En concreto, los elementos que alberga son de tipo estructura como se puede ver en el código 3.4.

```
1 struct ElementoOrdenado {
2     int fila;
3     int columna;
4     double valor;
5 };
6
7 typedef ElementoOrdenado elementoOrdenado;
```

Código 3.4: Estructura matriz ordenada de distancias

La finalidad de esta matriz es poder alterar el orden de las distancias de la matriz de distancias pero sin perder la información que provee la distancia, tal como la fila y la columna a la que pertenece. Este vector almacena el mismo número de elementos que el vector de distancias, incluidos los elementos de la diagonal con la suma de distancias de cada fila. Puede parecer que estamos duplicando información con respecto a la matriz de distancias pero la función que desempeña esta matriz no puede ser ofrecida eficientemente por la matriz de distancias original.

En este algoritmo el vector de distancias ordenadas se utiliza para ordenar las distancias de cada fila en orden descendente, es decir, de mayor a menor distancia de un elemento i (fila) con respecto a otro j (columna) siendo $i \leq j$. De esta forma nos aseguramos que todos los elementos posteriores a uno dado en la fila sean iguales o menores en sus distancias. El programa también permite su ordenación en orden ascendente o incluso no ser ordenado y mantener el mismo orden que el vector de distancias original.

Veremos la forma en que se utiliza este vector más adelante en 3.1.2.1 cuando se comente la forma en que se generan las soluciones del algoritmo.

La siguiente estructura de datos a comentar es la del **vector de suma de distancias**, que no es más que un vector donde se guardan las sumas de distancias de cada fila i en un orden determinado. La estructura utilizada para sus elementos es la del código 3.5. Como se puede observar es una estructura simple donde, al igual que en el vector de distancias ordenadas, se tiene que guardar la información del elemento al que pertenece la suma de distancias ya que su orden varía según su ordenación.

```
1  struct SumaDistancias_Estructura {
2      int nodo;
3      double sumaDistancias;
4  };
5
6  typedef SumaDistancias_Estructura elementoSumaDistancias;
```

Código 3.5: Estructura del vector de distancias

En este algoritmo se utiliza la ordenación en orden descendente para los elementos en base al valor de la suma de distancias del nodo con respecto a los demás elementos del problema. También se permite su ordenación en orden ascendente, aleatorio o sin orden alguno (en este caso el orden sería el de los nodos en orden ascendente).

Veremos su utilidad más adelante cuando se vea el proceso de mejora de soluciones en 3.1.3.

Las siguientes estructuras de datos utilizadas son las básicas para albergar los nodos que están incluidos en la solución y para saber que nodos están incluidos en la solución y cuales no además de otra información. La estructura de elementos del **vector solución** y del **vector de nodos** se muestran en el código 3.6.

```
1  struct ElementoSolucion_Estructura {
2      int nodo;
3      double aportacionSol;
4  };
5
6  typedef ElementoSolucion_Estructura elementoSolucion;
7
8  struct ElementoEnSolucion_Estructura {
9      bool enSolucion;
10     int vecesEnSolucion;
11     float ponderacion;
12 };
13
14 typedef ElementoEnSolucion_Estructura elementoEnSolucion;
```

Código 3.6: Estructura del vector solución y de nodos

La primera de las estructuras define los elementos pertenecientes al **vector solución** y almacena el nodo que está incluido en la solución y su aportación a la solución, o lo que es lo mismo, la suma de distancias con respecto a los demás elementos pertenecientes a la solución según la ecuación

$$aportacionSol(i) = \sum_{j=0}^{n-1} d_{ij} \quad i,j \in Solucion, \quad i \neq j \tag{3.6}$$

Este último valor será bastante útil para recalcular eficientemente el valor de la solución (véase 3.1.4) cuando se inserta o elimina un elemento de la solución como veremos más adelante. Hay que comentar que este vector tiene un tamaño de $m+1$ elementos, donde el elemento extra sirve para almacenar el número de elementos componentes de la solución, principalmente útil cuando se maneja una solución parcial, es decir, donde la solución no contiene los m elementos necesarios para que sea completa y válida, y que proporciona una forma eficiente de consultar el número de elementos en la solución.

La segunda estructura, el **vector de nodos**, además de decirnos rápidamente si un elemento está incluido en la solución actual, nos da información sobre cuantas veces ese elemento ha sido incluido en la solución además de un factor corrector ("ponderacion") a la hora de evaluar su posible inclusión en la solución. Este último campo podría recordar a una especie de "movimiento tabu" como en la metaheurística "Tabu" vista en 2.3.3.

La última estructura de datos que veremos será la del **conjunto de soluciones**. Este será un vector de elementos con la estructura mostrada en el código 3.7.

```
1 struct Elemento_Conjunto_Soluciones_Estructura {
2     elementoSolucion *solucion;
3     double valorSolucion;
4     long long tiempoInclusion; //Con respecto al inicio del algoritmo MDP
5                                //(sin incluir la fase de preparacion).
6 };
7
8 typedef Elemento_Conjunto_Soluciones_Estructura elementoConjuntoSoluciones;
```

Código 3.7: Estructura del conjunto de soluciones

Cada elemento del conjunto de soluciones almacenará un elemento **vector solución** correspondiente a la solución almacenada, el valor de la solución y el tiempo transcurrido desde el inicio del algoritmo en que se añade al conjunto de soluciones dicha solución. El conjunto de soluciones se crea de forma dinámica en base al tamaño especificado para el mismo mediante la configuración del algoritmo y, como vimos en 3.1, tiene su importancia en el mecanismo de reinicio de soluciones y es parte de la información devuelta por el algoritmo **GRASP_M**.

No se comentarán las estructuras de datos necesarias para la ejecución de "**PR**" pues no son esenciales para la ejecución del algoritmo. Para más información referirse al código fuente de la aplicación.

3.1.2. El generador de soluciones iniciales

El generador de soluciones es un elemento clave del algoritmo **GRASP_M** porque provee las soluciones susceptibles de ser mejoradas y debe hacerlo de tal forma que sea

rápido y provea soluciones diversas a la vez que susceptibles de ser buenas para la obtención de un buen valor para el problema *MDP*.

No se especificará su pseudocódigo porque, básicamente, lo que se realiza es encontrar 1 o 2 candidatos (como veremos en el siguiente apartado) a ser insertados en la solución actual y sustituir el/los elemento/s de la solución actual que se elijan en caso de que esta se encuentre ya completa.

Comentar que el proceso de generar soluciones, al insertar el/los nuevo/s elemento/s en la solución, ya ha calculado de manera eficiente (ver 3.1.4) el valor de la nueva solución generada como medida de ahorro de tiempo que supondría calcular el valor de la solución en base a todos los elementos de la misma un número de veces elevado.

No se pretende con este proceso llegar a soluciones óptimas sino que, en conjunción con el proceso de mejora y la reasignación de la solución generada actual a la mejor solución encontrada hasta el momento por la mejora, obtener buenas soluciones rápidamente.

La verdadera funcionalidad reside en los procesos para seleccionar los candidatos a entrar en la solución y para ser sustituidos que serán analizados en más profundidad a continuación.

3.1.2.1. Selección de candidatos a ingresar en la solución

Este es el proceso con el que se buscará 1 o 2 candidatos a agregar a la solución inicial actual. Este/os candidato/s se buscarán en la matriz de distancias ordenadas (la estructura de datos anteriormente comentada en 3.1.1.3). El hecho de que puedan seleccionarse 1 o 2 candidatos se debe a que se seleccionará un elemento de la matriz ordenada que, como sabemos, contiene información de la fila y la columna de una distancia del problema. Si tanto la fila como la columna de la distancia seleccionada no se encuentran en la solución se seleccionarán los dos elementos para entrar en la nueva solución generada. En caso contrario, o tan sólo el elemento de la fila o de la columna de la distancia seleccionada serán escogidos para su entrada en la nueva solución generada. El objetivo es que la distancia seleccionada esté incluida (tanto el elemento i de la fila como el j de la columna estén incluidos en la solución) en el cálculo de la nueva solución generada. El pseudocódigo del proceso se puede ver en el código 3.8.

El proceso consta de 3 métodos para la selección de los elementos a incluir en la solución: en base a la distancia entre dos elementos cualquiera del problema (Distancias), en base a la suma de distancias de un elemento i del problema con respecto a los demás elementos (SumaDistancias) y un híbrido entre los dos (Aleatorio) que es el utilizado en esta implementación del algoritmo **GRASP_M**.

Por sí solos, los tres métodos de selección tienen un claro afán por encontrar las mayores distancias entre elementos del problema para ser incluidas en la solución, una opción bastante factible en principio. Pero, por sí solo, este enfoque no es suficiente para encontrar las mejores soluciones al problema *MDP*. Es por ello que deben introducirse factores correctores para favorecer la diversidad de soluciones, es decir, una mayor y mejor exploración del espacio de soluciones.

El **primer factor corrector** es la propia aleatorización del método de selección. Con esta aleatorización se busca variar el criterio de selección y favorecer la diversidad.

```
Candidato = Vacio;
SI (ModoSeleccion == Aleatorio)
    ModoSeleccion = ModoAleatorio(Distancias,SumaDistancias);
SI (ModoSeleccion == Distancias) {
    PARA CADA i desde 0..(n-2) {
        NivelProfundidad = AleatorioEntre(0,n-i-1);
        PARA CADA j desde (i+1+NivelProdunfidad)..(n-1) {
            DistanciaActual = MatrizOrdenada[i,j];
            SI (CriterioSeleccionDistancias(DistanciaActual))
                Candidato = DistanciaActual;
            SI NO
                ActualizarPonderacion(DistanciaActual,Candidato);
        }
    }
    ActualizarPonderacion(Candidato);
} SI NO SI (ModoSeleccion == SumaDistancias) {
    PARA CADA i desde 0..(n-1)
        SumaDistanciaActual = MatrizOrdenada[i,i];
        SI (CriterioSeleccionSumaDistancias(SumaDistanciaActual)
            Candidato = SumaDistanciaActual;
        SI NO
            ActualizarPonderacion(SumaDistanciaActual,Candidato);
    }
    ActualizarPonderacion(Candidato);
}
Devuelve (Candidato);
```

Código 3.8: Pseudocódigo del proceso de búsqueda de candidatos a agregar a la solución

El **segundo factor corrector** pertenece al método de selección en base a distancias. Consiste en aleatorizar la columna de inspección de inicio para cada elemento del problema, excepto el último que no tiene valores de distancia en la matriz, desde donde se intentará obtener la mayor distancia entre elementos posible. La fórmula para realizar esta aleatorización para un elemento i del problema es la siguiente:

$$nivelProfundidad = (rand\,()) \;\%\; (n - i - 1)$$

De esta forma, para cada elemento i, se busca una distancia máxima con respecto a otro elemento j mayor que i tal que el inicio de la búsqueda comienza a partir de un elemento j aleatorio. Cuando se encuentre dicha distancia y o bien i o j no pertenezcan a la solución actual y sea la mayor distancia encontrada para cualquier elemento i (fila) se devuelve/n como candidato/s i y/o j según pertenezcan o no a la solución actual.

Es evidente que se recorren todos los elementos (filas i) del problema en la búsqueda de estos candidatos, pero como la matriz de distancias está ordenada en orden descendente se pueden obviar un gran número de columnas de la matriz, pues una vez encontrada una distancia menor o igual a la anterior que pueda ser candidata para una fila i no hace falta seguir analizando más columnas j para esa misma fila. De este modo este segundo factor corrector es eficiente y no ralentiza la generación de nueva soluciones.

El **tercer factor corrector** es una ponderación aplicada a cada elemento del problema. Este factor de ponderación está asociado a cada elemento independientemente, luego habrá n ponderaciones, una por elemento del problema. Este valor de ponderación se almacena en el vector de nodos, donde también se guarda si un elemento está incluido o no en la solución (véase 3.6).

El funcionamiento de esta ponderación es sencillo y puede recordar de algún modo a la lista de movimientos tabú utilizada en la metaheurística *TABU* (2.3.3).

Se trata de ponderar los valores de las distancias en la matriz de distancias ordenadas tanto por el valor de ponderación de la fila como de la columna (siempre que no pertenezcan a la solución la fila y/o columna pues ese elemento no entrará en la solución) a la que pertenece la distancia. Si alguno de estos valores ponderados merece ser candidato a entrar en la solución entonces se le hace candidato temporal.

Si, a causa de la ponderación, un elemento (fila y/o columna) que debería considerarse como candidato temporal (si no se le hubiera aplicado la ponderación) no lo es, el valor de la ponderación para ese elemento se incrementa (suma) en un valor de 0,1 sin sobrepasar nunca el valor de 1 máximo. De este modo el peso de la distancia real se va recuperando progresivamente.

Si un/os elemento/s (fila y/o columna) es finalmente considerado como candidato final a entrar en la solución, el valor de ponderación se decrementa en una décima parte mediante el producto de su ponderación por 0,1, previniendo su posible inclusión en, por lo menos, unas cuantas iteraciones siguientes (recuérdese que el valor de la ponderación para el elemento puede ir recuperándose progresivamente si es obviado a causa de la ponderación).

Este mecanismo se aplica igual en los dos métodos de selección existentes y está claro que los valores de reducción y recuperación de la ponderación pueden ser parametrizables y susceptibles de experimentación.

Veamos un ejemplo rápido y sencillo del sistema de ponderación. Tenemos un candidato temporal con una distancia igual a 100 y evaluamos para ser posible sustituto del candidato temporal una distancia de 120 y ponderación igual a 0,5, por ejemplo, para la fila i (en un pasado posiblemente no muy lejano el elemento i correspondiente a la fila de la distancia se incluyó en una solución generada pero actualmente no lo está). El posible sustituto sólo sustituiría al candidato temporal actual si $(120 * 0{,}5) > 100$ o si se diera el mismo caso para la columna j (j no debería encontrarse en la actual solución generada) de la distancia. Como se puede observar no lo sustituirá porque $(120 * 0{,}5) = 60 < 100$ pero en realidad la distancia real del posible sustituto es mayor que la del candidato temporal actual. Por tanto, la ponderación de 0,5 del elemento correspondiente a la fila se incrementará (recuperará) en un valor de 0,1 pasando a ser 0,6. En una próxima ocasión se utilizará esta nueva ponderación.

Supongamos ahora que esa distancia de 120 hubiera sido elegida como candidata final a ser insertada en la solución y, por ejemplo, el elemento i correspondiente a la fila no pertenece a la solución y tiene una ponderación de 1. Entonces se aplica un factor reductor de 0,1 a la ponderación del elemento i correspondiente a la fila de la distancia quedando la ponderación igual a $1 * 0{,}1 = 0{,}1$.

3.1.2.2. Selección de candidato a eliminar de la solución

En este proceso, cuando la solución está completa, se elige un elemento perteneciente a la solución para ser sustituido por el candidato a entrar a la nueva solución generada. En caso de no estar completa la solución se inserta el elemento directamente en ella. Su pseudocódigo se muestra en el código 3.9.

```
1  Candidato = 0; //Primer elemento de la solucion.
2  CandidatoAportacion = 0;
3
4  SI (MetodoSeleccion == Combinado)
5      MetodoSeleccion = SeleccionAleatoriaMetodoEntre(Valores,Aleatorio);
6
7  SI (MetodoSeleccion == Valores) {
8      Candidato = SeleccionaMenorElementoSolucionEnSumaDistancias(Solucion);
9      CandidatoAportacion = SeleccionaMenorElementoSolucionEnAporteASolucion(Solucion);
10     SI (Candidato != CandidatoAportacion)
11         Candidato = SeleccionAleatoriaEntre(Candidato,CandidatoAportacion);
12 }
13
14 SI (MetodoSeleccion == Aleatorio)
15     Candidato = SeleccionarElementoAleatorioValidoDeSolucion(Solucion);
16
17 Devolver(Candidato);
```

Código 3.9: Pseudocódigo del proceso de selección de candidatos a eliminar de solución

Se habilitan tres métodos para la selección del candidato a ser sustituido en la solución actual:

1. **Valores** selecciona un candidato en base a dos criterios: el primero selecciona un candidato perteneciente a la solución actual en base a la menor suma de distancias con respecto a todos los demás elementos del problema; el segundo selecciona un candidato en base a su menor aportación (suma de distancias del candidato con respecto a los demás elementos pertenecientes a la solución) a la solución actual. Si hay discrepancia entre los distintos candidatos seleccionados se realiza una selección aleatoria entre ellos.
2. **Aleatorio**, como su propio nombre indica, selecciona aleatoriamente como candidato un elemento válido de la solución, sin ningún otro criterio.
3. **Combinado** realiza una selección aleatoria entre los dos métodos anteriores.

Se establece por defecto el método de selección **Combinado** para la selección del elemento a salir de la aplicación. Si se quiere modificar el método hay que hacerlo mediante modificación del código fuente (no se encuentra parametrizado actualmente).

3.1.3. Mejora de soluciones

El proceso de mejora de soluciones es la parte fundamental del algoritmo pues se encarga de mejorar las soluciones de forma que se obtengan resultados buenos para la función objetivo del problema *MDP*. Su pseudocódigo se muestra en el código 3.10.

De entrada se inspeccionan todos los elementos del problema que no pertenezcan a la solución (línea 8) en un orden proporcionado por el **vector de distancias**, que si recordamos estaba compuesto por los elementos del problema ordenados, por defecto, en orden descendente según su suma de distancias con respecto a los demás elementos del problema (consultar 3.1.1.3).

En un segundo paso (línea 10) se compara la aportación a la solución que tendría el elemento i, con respecto a todos los elementos pertenecientes a la solución actual, con la

```
1  tg = 0;
2  MejorSolucionGlobal = solucion;
3  MIENTRAS (MejorSolucion(t+1) > MejorSolucion(t) {
4      MejoraSolucionSinElementos[m];
5      MenorAportacionSol = 0;
6      t = 0;
7      MejorSolucion = X(t) = SolucionActual;
8      PARA CADA i NoPertenece(X(t)) y EnOrden(VectorSumaDistancias) {
9          MejorSolucionSinElementos = Vacio;
10         SI (AportacionSol(i) > MenorAportacionSol) {//Criba de elementos a no tratar.
11             PARA CADA j Pertenece(X(t))
12                 SI (ValorSolucion(X(t),i,j) > ValorSolucion(X(t))
13                     IncluirEn(MejorSolucionSinElementos,j);
14             SI (MejorSolucionSinElementos != Vacio)
15                 X(t+1) =            AplicarCambio(X(t),i,SeleccionAleatoria(
                       MejorSolucionSinElementos);
16         }
17
18         SI (X(t+1) > MejorSolucion)
19             MejorSolucion = X(t+1);
20         SI (X(t) != X(t+1))
21             MenorAportacionSol = CalcularMenorAportacionSol(X(t+1));
22         t = t + 1;
23     }
24
25     SI (MejorSolucion(tg+1) > MejorSolucion(tg))
26         MejorSolucionGlobal = MejorSolucion(tg+1);
27     tg = tg + 1;
28 }
29
30 Devolver(MejorSolucion(tg));
```

Código 3.10: Pseudocódigo del proceso de mejora de soluciones

menor aportación que un elemento de la solución aporta a la misma, es decir, la aportación que realizaría i a la solución si se añadiera a la solución sin eliminar ningún elemento de la misma. Si dicha aportación de i es menor no vale la pena considerarlo porque nunca se podrá obtener una mejor solución en cuanto a valor de la función objetivo y, por tanto, sería un desperdicio de recursos tratarlo. De esta forma se criban bastantes elementos del problema y se acelera el proceso de mejora. A continuación se muestra la formulación de esta comparación:

$$Aportacion(i) = \sum_{j=0}^{m-1} d(i,j) < min\left(\sum_{k=0}^{m-1} d(j,k)\right) \quad k \neq j, \quad \forall j \in Solucion \tag{3.7}$$

que si se cumple no se continúa con el proceso de intentar mejorar la solución con el elemento i (no perteneciente a la solución) del problema.

Posteriormente (líneas 11 a 13), se realiza la comprobación del valor de la solución si se sustituyera cada elemento j de la solución con i. Si el valor de la solución mejora con el intercambio por i de un determinado elemento j de la solución, se guarda j como un posible cambio por i a la misma.

Si hay varios posibles cambios que mejoren la solución actual introduciendo i en la solución (líneas 14 y 15), la solución actual es la evolución de la solución inicial con los cambios realizados donde se producen mejoras con respecto a la solución inicial aporta-

da. Se decide aleatoriamente entre los distintos posibles cambios para realizar el cambio de la solución actual.

La nueva solución mejorada, si es mejor que la mejor solución aportada hasta el momento por el proceso de mejora, se convierte en la mejor solución encontrada que al final será devuelta por el proceso (líneas 18 y 19).

Si se ha producido un cambio en la solución actual se deben recalcular los valores de aportación de los elementos a la solución (de forma eficiente) y guardar el elemento que menor valor aporte a la solución (líneas 20 y 21) para poder cribar elementos siguientes a comprobar para entrar en la solución actual modificada.

Este proceso de mejora es iterativo (línea 3), es decir, se realiza consecutivamente mientras haya habido mejora de la solución aportada al proceso. Si no se produce mejora de la solución aportada se termina el proceso de mejora y se devuelve el valor de la mejor solución encontrada.

Como ya se ha dicho en varias ocasiones en este documento este es un **proceso costoso a nivel de procesamiento**. Se puede observar como se realiza un bucle anidado de orden algorítmico $O(N \cdot M)$, aunque con la criba de elementos se reduce bastante el tiempo de procesamiento, y se tienen que realizar diversos cálculos de aportes a la solución con los cambios a la solución efectuados y con la toma en consideración de i como posible cambio, lo que tiene un orden algorítmico de M. Además, el proceso, al ser iterativo mientras se produzcan mejoras, se puede realizar varias veces por cada solucion generada a mejorar. Debido a esto hay que afinar mucho la aplicación de procesos de mejora para que, sin ser exhaustiva, cumpla con su cometido efectivamente.

A favor de este proceso de mejora podemos decir que suele dar soluciones bastante buenas en cuanto al valor de la función objetivo buscado.

3.1.4. Cálculo eficiente del valor de las soluciones

Una solución tiene un valor V en cuanto a la función objetivo y cada elemento i perteneciente a la solución aporta un determinado valor $Aportacion(i)$ con respecto a los demás elementos pertenecientes a la solución:

$$Aportacion(i) = \sum_{j=0}^{m-1} d(i,j) \quad \forall j \neq i \quad j, i \in Solucion \tag{3.8}$$

Esta aportación se calcula para cada elemento perteneciente a la solución cada vez que varíe la solución por la inclusión de algún nuevo elemento a la misma (recuérdese su almacenamiento en el vector solución en 3.1.1.3).

Cada vez que se inserta un nuevo elemento i por uno j en la solución o se quiere comprobar la aportación de un nuevo elemento i que sustituye a un elemento j perteneciente a la solución, el nuevo valor de la solución se calcula de la siguiente forma:

$$Solucion(i) = Solucion(j) + Aportacion(i) - Aportacion(j) \tag{3.9}$$

donde:

$$Aportacion(i) - Aportacion(j) = \sum_{k=0}^{m-1} (d(i,k) - d(j,k)) \quad k \in Solucion \quad i,j \neq k \tag{3.10}$$

Por tanto se obtiene el incremento/decremento de la acción de sustituir i por j en la solución que sumado al valor de la solución anterior con el elemento j incluido nos da el valor de la solución con i incluido y j excluido.

El mantenimiento de la aportación a la solución de un nuevo elemento i incluido conlleva un coste de proceso menor que recalcular el valor de la solución cada vez que se sustituye un nuevo elemento, que sería:

$$ValorSolucion = \sum_{i=0}^{m-1} \sum_{j=i+1}^{m-1} d(i,j) \tag{3.11}$$

donde el orden de calcular el valor de la aportación a la solución de un elemento i para su uso en 3.9 sería M y el de calcular el valor de la solución según la fórmula 3.11 sería de orden $O\left(M^2\right)$. Esta operación de recalcular el valor de la solución tiene que realizarse muchas veces (sobre todo en la comprobación de si se obtiene una mejora en la solución si se realizara un cambio), luego se consigue una mejora significativa en el rendimiento del algoritmo.

Tras realizar un cambio a la solución, introduciendo un elemento i y extrayendo un elemento j, hay que recalcular las aportaciones a la solución de los demás elementos k de la solución que ya se encontraban en ella de acuerdo a la siguiente fórmula:

$$Aportacion(k) = Aportacion(k) + d(i,k) - d(j,k) \quad \forall k \neq i \tag{3.12}$$

es decir, disminuir la aportación a la solución de k en la distancia aportada por j y aumentar en la distancia aportada por i. Este recáclulo de las aportaciones es esencial para el buen funcionamiento del sistema de cálculo de soluciones. El orden algorítmico de esta operación es $O\left(M\right)$.

3.2. Resultados experimentales

En esta parte llevaremos a cabo diversos experimentos para comprobar tanto la optimización de los parámetros referentes al algoritmo como de la bondad en cuanto a soluciones aportadas y tiempo de ejecución empleado del mismo.

El entorno en el que se han realizado todos los experimentos es una máquina virtual Linux en Virtual Box v4.3.26 con Ubuntu Server v14.04.2 instalado como sistema operativo invitado con la instalación estándar y sin servicios adicionales (Servidor FTP, Servidor Correo…) activados/instalados. El hardware sobre el que corre la máquina virtual es un AMD Athlon II X3 435 de 3 núcleos con 2 GB de RAM y la máquina virtual tiene asignados 2 núcleos al 100 % y 770 MB RAM con el soporte de virtualización VT-X/AMD-V activado. Se ha elegido la versión Server de Ubuntu porque no viene instalado por defecto un entorno gráfico y, por tanto, se trabaja desde consola y, de esta forma, la interferencia con la realización de los experimentos es mínima.

El algoritmo no se ha implementado con ningún tipo de pararelismo (sin hilos de ejecución) salvo el que el propio Sistema Operativo pueda aportar.

3.2.1. Elección de parámetros del algoritmo

Para la experimentación de la elección de parámetros óptimos del algoritmo nos basaremos en pruebas realizadas sobre los conjuntos de datos de referencia *"GDK-c"* y *"MDG-a"* provenientes de [33].

La elección de estos conjuntos se decide porque tienen un valor representativo para la realización de pruebas, sobre todo en tamaños del problema. El conjunto *GDK-c* contiene 20 matrices de distancias de tamaño $n = 500$ y $m = 50$ y *MDG-a* (utilizadas en [37]) contiene 40 matrices de distancias, 20 de ellas con tamaño $n = 500$ y $m = 50$ y las otras 20 con tamaño $n = 2000$ y $m = 200$. Por razones de homogenización de los resultados obtenidos las matrices de distancias se han dividido en 3 grupos: *GKD-c*, *MDG-a I* y *MDG-a II*.

La **metodología** a seguir en este experimento será la de comparar una configuración por defecto predeterminada de los parámetros más significativos de funcionamiento del algoritmo con variaciones en estos parámetros uno a uno para comprobar si estos cambios mejoran o empeoran las prestaciones del algoritmo. Posteriormente, se realizará un análisis de los datos obtenidos por las distintas ejecuciones del algoritmo para escoger una configuración óptima de los parámetros de configuración del mismo.

Por ser la elección de una buena configuración de los parámetros del algoritmo esencial para posteriores pruebas, en este experimento se ha escogido un conjunto de datos más amplio (60 matrices de distancias) del que se escogerá para posteriores experimentos. Debido a la falta de tiempo no se puede realizar la experimentación sobre todo el conjunto completo de matrices de distancias de referencia que se encuentra en el *Proyecto Optsicom* [33].

En el experimento que nos ocupa se han realizado por cada configuración de parámetros 5 ejecuciones del algoritmo sobre cada matriz de distancias en sendas tandas de 30 y 100 segundos de tiempo límite de ejecución por matriz y ejecución.

La configuración de parámetros (véase 3.1.1.2) por defecto del algoritmo es la siguiente:

- **MDP_NSC** = 5. Capacidad del conjunto de soluciones.
- **MDP_flags** = 179 (MDP_FSDO + MDP_FSDOD + MDP_FDOO + MDP_FDOOD + MDP_FEPS). Banderas de configuración por defecto (véase código 3.3).
- **MDP_MISM** = 50. Número máximo de iteraciones consecutivas sin conseguir mejorar, tras aplicar el proceso de mejora a la solución generada actual, la mejor solución en la actual ejecución del algoritmo antes de proceder a su rearranque.
- **MDP_PM** = 0,05. Porcentaje (expresado en tanto por uno) mínimo de aplicaciones del proceso de mejora con respecto al número total de soluciones generadas por el algoritmo.

- **MDP_PREC** = 6. Precisión para comparación de cantidades tipo "double" en la sexta cifra decimal.

y las distintas variaciones de parámetros de configuración (una a una) con las que se han efectuado pruebas son:

- **MDP_MISM** = 100.
- **MDP_MISM** = 200.
- **MDP_FDOO** = desactivado. Desactivada la ordenación de la matriz de distancias.
- **MDP_FEPS** = desactivado. Desactivado el forzado de la aplicación del proceso de mejora a las soluciones generadas en las que aparece un elemento del problema por primera vez (desde el último arranque del algortimo).
- **MDP_PM** = 0,02.
- **MDP_PM** = 0,10.
- **MDP_PM** = 0,20.

En las tablas 3.1 y 3.2 se muestran los resultados de las distintas ejecuciones del algoritmo en función de los parámetros. En ellas, en la primera columna se muestra la configuración del algoritmo para la prueba en concreto, en la segunda se muestra el grupo de matrices que conforman el resultado, en la tercera el tiempo medio en milisegundos para un grupo de matrices de distancias en las que se ha encontrado la mejor solución y en la cuarta el "GAP" con respecto a la mejor solución aportada por [33] para las matrices evaluadas. El cálculo del "GAP" se ha realizado según la siguiente fórmula:

$$GAP = \frac{Sol_{Alg} - Sol_{Best}}{Sol_{Best}} \tag{3.13}$$

Cabe destacar que siguiendo la formulación de 3.13 para el cálculo del "GAP" se obtendrán valores negativos cuando la solución aportada por el algoritmo sea inferior a la mejor solución especificada en [33] y positivos cuando las soluciones aportadas por el algoritmo sean superiores. Por tanto, cuanto más se aproxime el valor de "GAP" a 0 o incluso sea positivo mejor será el comportamiento del algoritmo en cuanto a la resolución del problema *MDP*.

A continuación analizaremos los resultados de los experimentos mostrados en las tablas 3.1 y 3.2 e intentaremos obtener una configuración por defecto buena en general.

3.2.1.1. Parámetro MDP_MISM

En general, la configuración por defecto con **MDP_MISM** = 50 del algoritmo proporciona mejores resultados en la búsqueda de soluciones al problema *MDP* que el aumento de su valor a 100 o 200 para el conjunto de matrices seleccionadas.

El aumento del parámetro *MDP_MISM* significa, en realidad, realizar una intensificación en la búsqueda de una solución al problema MDP en cada arranque del algoritmo,

PARAMETROS	TIPO PROBLEMA	T. MEDIO SOL. (ms.)	GAP MEDIO
DEFECTO T = 30	GKD-A	12,84	-0,00000017
	MDG-A I	7642,14	-0,00024042
	MDG-A II	19290,76	-0,00125391
	TOTAL	8981,91	-0,00049817
DEFECTO T = 100	GKD-A	15,26	-0,00000017
	MDG-A I	20060,28	-0,00005913
	MDG-A II	56061,28	-0,00070135
	TOTAL	25378,94	-0,00025355
MISM 100 T= 30	GKD-A	13,36	-0,00000017
	MDG-A I	8247,74	-0,00016538
	MDG-A II	14990,08	-0,00180493
	TOTAL	7750,39	-0,00065682
MISM 100 T= 100	GKD-A	14,24	-0,00000017
	MDG-A I	18968,78	-0,00008262
	MDG-A II	60197,20	-0,00084294
	TOTAL	26393,41	-0,00030858
MISM 200 T= 30	GKD-A	11,56	-0,00000017
	MDG-A I	7570,34	-0,00013822
	MDG-A II	12220,46	-0,00173253
	TOTAL	6600,79	-0,00062364
MISM 200 T= 100	GKD-A	11,34	-0,00000017
	MDG-A I	25751,20	-0,00005330
	MDG-A II	49725,22	-0,00124251
	TOTAL	25162,59	-0,00043199
DISTANCIAS NO ORDENADAS T = 30	GKD-A	64,44	-0,00000017
	MDG-A I	9896,82	-0,00032526
	MDG-A II	20816,74	-0,00209987
	TOTAL	10259,33	-0,00080843
DISTANCIAS NO ORDENADAS T = 100	GKD-A	77,44	-0,00000017
	MDG-A I	24657,20	-0,00012477
	MDG-A II	62119,86	-0,00098853
	TOTAL	28951,50	-0,00037116

Tabla 3.1: Primer conjunto de experimentos de parámetros del algoritmo GRASP_M

PARAMETROS	TIPO PROBLEMA	T. MEDIO SOL. (ms.)	GAP MEDIO
NO FEPS T = 30	GKD-A	14,12	-0,00000017
	MDG-A I	8980,20	-0,00023996
	MDG-A II	17739,11	-0,00143937
	TOTAL	9195,92	-0,00058821
NO FEPS T = 100	GKD-A	11,20	-0,00000017
	MDG-A I	15949,16	-0,00007587
	MDG-A II	54189,52	-0,00060190
	TOTAL	23383,29	-0,00022598
PM 0.02 T = 30	GKD-A	10,96	-0,00000017
	MDG-A I	7810,02	-0,00023076
	MDG-A II	18822,00	-0,00118208
	TOTAL	8880,99	-0,00047100
PM 0.02 T = 100	GKD-A	14,08	-0,00000017
	MDG-A I	24171,24	-0,00004499
	MDG-A II	57091,04	-0,00081008
	TOTAL	27092,12	-0,00028508
PM 0.10 T = 30	GKD-A	12,04	-0,00000017
	MDG-A I	8749,04	-0,00015883
	MDG-A II	17801,84	-0,00151111
	TOTAL	8854,31	-0,00055670
PM 0.10 T = 100	GKD-A	11,40	-0,00000017
	MDG-A I	19116,12	-0,00008675
	MDG-A II	54355,94	-0,00067180
	TOTAL	24494,49	-0,00025291
PM 0.20 T = 30	GKD-A	10,02	-0,00000017
	MDG-A I	6213,98	-0,00005750
	MDG-A II	18860,96	-0,00138307
	TOTAL	8361,65	-0,00048025
PM 0.20 T = 100	GKD-A	12,40	-0,00000017
	MDG-A I	8713,32	-0,00001267
	MDG-A II	48271,64	-0,00089897
	TOTAL	18999,12	-0,00030394

Tabla 3.2: Segundo conjunto de experimentos de parámetros del algoritmo GRASP_M

lo que conlleva que, para un mismo tiempo límite del algoritmo, se realizarán menos rearranques del mismo a mayor valor del parámetro *MDP_MISM*.

La experimentación parece indicar que es preferible que el algoritmo tenga un mayor número de rearranques que intensificar la búsqueda de soluciones en cada rearranque del mismo. Es por ello que se ha escogido un valor por defecto de 50 para el parámetro para favorecer este hecho sin llegar a producirse rearranques muy frecuentes que no aporten soluciones lo bastante buenas.

Como observación, indicar que para los tamaños de problema más pequeños (MDG-A I, 500×50) sí se ha producido una leve mejora con respecto al valor del parámetro por defecto, pero no así para tamaños de problema más grandes (MDG-A II, 2000×200), donde se produce un empeoramiento mayor.

Con respecto a los tiempos medios en que se encuentran las mejores soluciones no se obtienen conclusiones evidentes puesto que para los tiempos límite menores ($T = 30$) parece ser que se reduce un poco este tiempo pero para tiempos límite mayores ($T = 100$) no se observa este hecho.

3.2.1.2. Parámetro MDP_FDOO (Distancias Ordenadas)

En cuanto al cambio en el parámetro **MDP_FDOO** (ordenamiento de la matriz de distancias) se observa que el no ordenar la matriz de distancias en orden descendente (valor por defecto) sino dejarla tal como se ha leído de la entrada provoca un empeoramiento significativo, mayor cuanto menor es el tiempo límite del algoritmo, de los resultados del algoritmo tanto en el valor de las soluciones obtenidas como en los tiempos medios en que se encuentran esas soluciones.

Por tanto, se considera mejor la opción, por defecto, de ordenar la matriz de distancias leída en orden descendente que dejarla sin ordenar, aunque esto conlleve un tiempo de procesamiento mayor (en función del número de distancias del problema) en la fase de preparación del algoritmo debido a tener que realizarse esta ordenación.

3.2.1.3. Parámetro MDP_FEPS

El parámetro **MDP_FEPS** incide en uno de los criterios para la aplicación del proceso de mejora a las soluciones generadas por el algoritmo. Si está activo implica aplicar el proceso de mejora a las soluciones generadas por el algoritmo siempre que se haya introducido un elemento en la solución generada por primera vez desde el último arranque del algoritmo.

Según los resultados experimentales, parece ser que la no aplicación de este criterio de aplicación del proceso de mejora produce mejores resultados, tanto en tiempo medio de obtención de las mejores soluciones como en el valor de las mismas, para tiempos límite del algoritmo mayores ($T = 100$), no habiendo mucha diferencia para tiempos menores ($T = 30$).

Este parámetro, al igual que *MDP_MISM*, produce un efecto de intensificación en la búsqueda de soluciones pues provoca un mayor número de aplicaciones del proceso de

mejora (por lo menos en las fases iniciales del algoritmo) sobre las soluciones generadas por el algoritmo. Parece ser que los resultados obtenidos mejoran con este parámetro desactivado para problemas del tipo más grande (MDG-A II, 2000×200) y tiempos límite mayores ($T = 100$). Esto podría ser debido a que, con el parámetro desactivado, se produce una mayor dispersión de las aplicaciones del proceso de mejora a lo largo de la exploración del espacio de soluciones.

Por tanto, a la hora de especificar este parámetro se tendrá en cuenta lo anteriormente expuesto. A falta de una experimentación más exhaustiva se dejará el parámetro *MDP_FEPS* desactivado en la configuración por defecto.

3.2.1.4. Parámetro MDP_PM

El parámetro **MDP_PM** (Porcentaje de Mejoras) del algoritmo indica el porcentaje mínimo de aplicaciones del proceso de mejora con respecto al número total de soluciones generadas por el mismo. El valor de este parámetro, por lo tanto, también incide en la mayor o menor intensificación en la búsqueda de soluciones del algoritmo (a mayor valor del parámetro más intensificación/aplicación del proceso de mejora a soluciones generadas).

La experimentación parece indicar que un valor del parámetro entre 0,02 y 0,10 (2 % y 10 % respectivamente) produce unos resultados mejores. El valor por defecto para *MDP_PM* se ha establecido a 0,05 (5 % de mejoras), un valor intermedio que parece proporcionar buenos resultados en general.

3.2.1.5. Conclusiones

En cuanto a las conclusiones obtenidas tras la experimentación con el cambio de los valores de los parámetros más importantes del algoritmo se puede observar como los parámetros definidos por defecto, previos a la experimentación, funcioban bastante bien en general, tanto en los tiempos medios de la obtención de las mejores soluciones como en el "GAP" medio sobre los conjuntos de matrices.

Prueba de esto es que, tras la experimentación, tan sólo se ha modificaco uno de estos parámetros por defecto. *MDP_FEPS* ha sido desactivado en la nueva configuración por defecto con la que se realizarán las pruebas de "Bondad" del algoritmo.

Añadir que existen otros parámetros y otras configuraciones (véase 3.1.1.2) que pueden afectar a los resultados obtenidos, pero una experimentación exhaustiva requeriría de unos tiempos de cómputo muy elevados. Más que en obtener los valores óptimos estamos interesados en analizar una metodología para la determinación de valores satisfactorios de estos parámetros. De los experimentos y las conclusiones anteriores podemos considerar que los parámetros del algoritmo estudiados son de los más importantes en relación a la bondad de la solución obenida.

Se propone para posibles trabajos posteriores el estudio más exhaustivo de los distintos parámetros del algoritmo.

3.2.2. Bondad (Fitness) del algoritmo

En este apartado realizaremos pruebas para comprobar la "**Bondad**" (*Fitness*) del algoritmo con un conjunto de matrices más variado que en la prueba para la elección de parámetros. El entorno de ejecución del algoritmo es el mismo que el utlizado en la sección 3.2.1 y los parámetros aplicados al algoritmo son los obtenidos como óptimos en la misma sección.

Decir que la *Bondad* del algoritmo se medirá en términos de "*GAP*" (véase fórmula 3.13) con respecto a las mejores soluciones aportadas por [33] para las matrices de distancias de mayor tamaño.

Por razones de no poder extenderse mucho en esta fase de experimentación, se han seleccionado representantes, al azar, de los distintos conjuntos disponibles en [33] cuyos tamaños de matriz sean superiores a $n = 500$, $m = 50$ puesto que para tamaños iguales o menores, con los tiempos límite de ejecución del algoritmo impuestos en esta prueba, los resultados experimentales suelen ser poco significativos porque se encuentran los valores óptimos sin gran dificultad.

Las matrices de distancias, seleccionadas al azar, con las que se realizarán las pruebas son las 8 siguientes:

1 **MDG-a_21_n2000_m200.txt**

2 **MDG-a_40_n2000_m200.txt**

3 **MDG-b_21_n2000_m200.txt**

4 **MDG-b_40_n2000_m200.txt**

5 **MDG-c_1_n3000_m300.txt**

6 **MDG-c_6_n3000_m400.txt**

7 **MDG-c_11_n3000_m500.txt**

8 **MDG-c_16_n3000_m600.txt**

Se ha intentado seleccionar un mínimo de 2 matrices por cada conjunto así como seleccionar un tamaño de problema y solución distinto de cada conjunto de matrices, como se ha dicho antes, de entre los tamaños de problema mayores.

Con respecto a los tiempos de ejecución límite del algoritmo se han seleccionado 100, 300 (5 minutos) y 900 (15 minutos) segundos, los cuales se consideran suficientes para ver la evolución del algoritmo con respecto al tiempo límite de su ejecución. Para cada matriz y tiempo límite se han realizado 5 ejecuciones distintas del algoritmo, obeniendo valores medios de los valores en función de estas ejecuciones. Los resultados de las pruebas se muestran en las tablas 3.3, 3.4 y 3.5 (100, 300 y 900 segundos respectivamente).

Se pueden observar en la tabla 3.3 los resultados obtenidos donde el *GAP* medio para el conjunto de matrices seleccionado y tiempo límite del algoritmo de 100 segundos es relativamente bueno, teniendo en consideración que el tamaño de las matrices tenidas en cuenta es bastante superior a aquel utilizado para la experimentación de los parámetros

PROBLEMA	TIEMPO MEDIO SOLUCION (ms.)	GAP MEDIO
MDG-a_21_n2000_m200.txt	51177,20	-0,00103099
MDG-a_40_n2000_m200.txt	50620,00	-0,00023645
MDG-b_21_n2000_m200.txt	48783,60	-0,00121625
MDG-b_40_n2000_m200.txt	41556,00	-0,00108715
MDG-c_1_n3000_m300.txt	47795,8	0,00008094
MDG-c_6_n3000_m400.txt	72505,4	-0,00174013
MDG-c_11_n3000_m500.txt	82397,8	-0,00046522
MDG-c_16_n3000_m600.txt	70075,2	0,00010084
TOTAL	58113,875	-0,00069930

Tabla 3.3: Fitness obtenido con $T = 100$

PROBLEMA	TIEMPO MEDIO SOLUCION (ms.)	GAP MEDIO
MDG-a_21_n2000_m200.txt	208052,40	-0,00061789
MDG-a_40_n2000_m200.txt	188161,60	-0,00014712
MDG-b_21_n2000_m200.txt	226191,20	-0,00097833
MDG-b_40_n2000_m200.txt	167246,60	-0,00066335
MDG-c_1_n3000_m300.txt	186416	0,00049852
MDG-c_6_n3000_m400.txt	83154,6	-0,00147510
MDG-c_11_n3000_m500.txt	248957,4	-0,00004959
MDG-c_16_n3000_m600.txt	183364	0,00029583
TOTAL	186442,975	-0,00039213

Tabla 3.4: Fitness obtenido con $T = 300$

PROBLEMA	TIEMPO MEDIO SOLUCION (ms.)	GAP MEDIO
MDG-a_21_n2000_m200.txt	666306,40	-0,00040785
MDG-a_40_n2000_m200.txt	741897,40	0,00018741
MDG-b_21_n2000_m200.txt	628067,60	-0,00065489
MDG-b_40_n2000_m200.txt	516584,40	-0,00056297
MDG-c_1_n3000_m300.txt	507611,6	0,00093653
MDG-c_6_n3000_m400.txt	340628,6	-0,00118294
MDG-c_11_n3000_m500.txt	574951	0,00006823
MDG-c_16_n3000_m600.txt	592137,8	0,00062640
TOTAL	571023,1	-0,00012376

Tabla 3.5: Fitness obtenido con $T = 900$

del algoritmo. Este *GAP* medio de $-0,00069930$ parece interesante y, a pesar de no ser un tiempo elevado, para el tamaño de los problemas tratados, incluso hay dos matrices (MDG-c_1_n3000_m300 y MDG-c_16_n3000_m600) en las que el *GAP* es positivo, lo que significa que se han encontrado soluciones mejores (en media) a las propuestas por [33].

En la tabla 3.4 se observa que al aumentar el tiempo límite del algoritmo a 300 segundos (5 minutos) el resultado del algoritmo mejora con un *GAP* medio de $-0,00039213$, una reducción de alrededor del 44 % en comparación con la ejecución con tiempo límite de 100 segundos. Los resultados mejoran para todas las matrices de distancias, lo que implica que el funcionamiento del algoritmo parece correcto ya que a mayor tiempo de procesamiento mejores resultados se obtienen.

Con respecto a los valores obtenidos para un tiempo de ejecución de 900 segundos (15 minutos), en la tabla 3.5 se observa que el *GAP* medio sigue reduciéndose, con un valor medio de $-0,00012376$, lo que supone alrededor de un 68 % de mejora con respecto a la ejecución con un límite de 300 segundos. A su vez, ahora hay más matrices de distancias tratadas (MDG-a_40_n2000_m200, MDG-c_1_n3000_m300, MDG-c_11_n3000_m500 y MDG-c_16_n3000_m600) en las que el *GAP* medio que se obtiene es positivo. Todo esto confirma que el funcionamiento del algoritmo parece correcto al mejorar con un mayor tiempo límite para encontrar soluciones al problema *MDP*.

Como curiosidad, según los resultados experimentales, cabe destacar el comportamiento del problema "MDG-c_6_n3000_m400" que presenta el *GAP* más elevado de entre todos los problemas tratados. Además, teniendo en cuenta los tiempos medios en los que se encuentra la mejor solución encontrada vemos como, para este problema, el valor es relativamente menor con respecto a los demás problemas, sobre todo para los tiempos límite mayores (300 y 900 segundos). El que este tiempo sea menor significa que hay un punto de la ejecución en que el algoritmo no encuentra mejores soluciones y que este punto es relativamente temprano. Esto podría indicar que pudiera haber ciertos problemas en los que al algoritmo *GRASP_M* le resultara más difícil encontrar las mejores soluciones.

3.3. Conclusión

Como **conclusión** general, y a falta de una experimentación más exhaustiva, se puede decir que el funcionamiento del algoritmo es correcto y, en general, obtiene unos resultados buenos siempre en comparación con los aportados por [33] y teniendo en cuenta que los tiempos de ejecución límite del algoritmo **GRASP_M** son menores a los utilizados por [33] (2 horas).

Se ha diseñado un algoritmo parametrizado, lo que permite experimentar variando los valores de los parámetros para obtener configuraciones de los parámetros con los que se obtienen buenas soluciones. Se ha mostrado una metodología de evaluación de la influencia de los parámetros en la bondad de la solución, realizando un análisis para problemas no demasiado grandes y contrastando para tamaños mayores que los valores seleccionados proporcionan buenas soluciones incluso en casos complejos, llegando a mejorar algunos de los resultados obtenidos en la base de problemas que se utiliza en los experimentos [33]. Se propone, para futuros trabajos, una experimentación con un número de matrices de distancia mayores, tiempos de ejecución límite mayores y otros conjuntos de datos distintos a los ofrecidos en esta base de problemas.

Capítulo 4

Un esquema metaheurístico parametrizado aplicado a la resolución de MDP

En este capítulo experimentamos con el esquema metaheurístico parametrizado introducido en [1], que permite la implementación de varias metaheurísticas distintas, basado en el uso de parámetros, utilizando un algoritmo común y aplicable a diversos tipos de problemas con tan sólo modificar las funciones básicas del esquema.

Posteriormente, realizaremos una serie de experimentos para analizar el funcionamiento de este esquema metaheurístico parametrizado aplicado a la resolución del problema *MDP* intentando obtener conclusiones en cuanto a su rendimiento e influencia de los parámetros en este objetivo. En estos experimentos, al igual que para el algoritmo desarrollado en el capítulo anterior, se analiza la influencia de los parámetros en la bondad de la solución obtenida.

4.1. El esquema metaheurístico parametrizado (EMP)

Existen multitud de problemas de optimización, principalmente, de tipo combinatorio que necesitan ser resueltos con eficiencia en un tiempo factible. Para la resolución de este tipo de problemas eficientemente no queda otra opción, generalmente, que el uso de heurísticas y metaheurísticas.

La justificación de la utilización de *EMPs* es que, generalmente, la adaptación de una metaheurística a cada tipo de problema necesita de un gran esfuerzo en la implementación y optimización de la misma para la obtención de buenos resultados. Además, algunas metaheurísticas no son aplicables o adecuadas según el tipo de problema al que se quieran aplicar. El *EMP* permite afrontar varios de los problemas que las metaheurísticas presentan en cuanto su adaptación a varios tipos de problemas. Concretamente, se intenta incrementar el nivel de genericidad de las metaheurísticas sin pérdida de rendimiento haciendo posible su adaptación a diversos tipos de problemas.

Con la introducción de los "*Parámetros Transicionales*", y gracias al nivel de abstración del *EMP*, se puede no sólo conseguir la aplicación de metaheurísticas puras sino que tam-

```
Inicializar(S, ParamIni)
while (not ConficionFin(S, ParamFin)) {
    SS = Seleccionar(S, ParamSel)
    SS1 = Combinar(SS, ParamCom)
    SS2 = Mejorar(SS1, ParamMej)
    S = Incluir(SS2, ParamInc)
}
```

Código 4.1: Esquema Metaheurístico Parametrizado (EMP)

bién se puede conseguir la obtención de metaheurísticas híbridas entre ellas e, incluso, la obtención de nuevas metaheurísticas, con tan sólo la especificación de los valores de los *parámetros transicionales*.

En [1] se especifican estos parámetros y su significado, además de aplicar el *EMP* a tres tipos de problemas distintos y analizar la influencia de los parámetros en la ejecución del *EMP*. En este *EMP* se ofrece la posibilidad de obtener las metaheurísticas *GRASP*, *Scatter Search* (SS) y *Genetic Algorithm* (GA), así como distintas variantes basadas en la especificiación de los parámetros. Su algoritmo se muestra en el listado de código 4.1.

4.2. Pruebas experimentales

En este apartado realizaremos pruebas experimentales para intentar obtener conclusiones sobre la aplicación del *EMP* a un problema determinado, en este caso al *MDP*.

La metodología a seguir será la de realizar ejecuciones sobre cada uno de los problemas *MDP* seleccionados en una elección de problemas (provinientes de [33]) significativos en cuanto a tamaño de problema/solución, sin escoger tamaños de problema grandes pues la ejecución del *EMP* puede llegar a tardar bastante tiempo con este tipo de problemas y no se desea alargar la duración del experimento en exceso. Para cada uno de estos problemas se realizarán 5 ejecuciones, para cada uno de los intervalos de tiempo máximo de 30 y 100 segundos, con cada una de las configuraciones de parámetros de referencia para distintas metaheurísticas e hibridaciones aportadas por [1].

En concreto, los problemas seleccionados para su experimentación son:

- GKD-a_50_n15_m12.txt
- GKD-a_51_n30_m6.txt
- GKD-a_75_n30_m24.txt
- SOM-a_20_n50_m15.txt
- SOM-a_21_n100_m10.txt
- SOM-a_50_n150_m45.txt

Las configuraciones referencia de los parámetros para el *EMP* empleadas en las pruebas, según se especifican en [1], para distintas metaheurísticas puras e hibridaciones son las que se muestran en la tabla 4.1.

	GRASP	GA	SS	GRASP+GA	GRASP+SS	GA+SS	GRASP+GA+SS
INEIni	100	100	100	100	100	100	100
FNEIni	10	100	20	100	20	50	50
PEIIni	100	0	100	100	100	100	100
IIEIni	10	0	10	10	10	5	10
MNIEnd	0	100	100	100	100	100	100
NIREnd	2	10	10	10	10	10	10
NBESel	10	100	10	100	10	38	38
NWESel	0	0	10	0	10	12	12
PBBCom	5	50	45	50	45	703	703
PBWCom	0	0	100	0	100	456	456
PWWCom	0	0	45	0	45	66	66
PEIImp	0	0	100	0	50	50	50
IIEImp	0	0	10	0	10	5	5
PDIImp	0	5	0	5	0	50	50
IIDImp	0	0	0	0	0	0	0
NEBInC	10	100	10	100	10	37	37

Tabla 4.1: Valor de los parámetros transicionales en EMP para distintas metaheurísticas básicas e hibridaciones

El entorno donde se han realizado las pruebas es el mismo que en la experimentación con el algoritmo *GRASP_M* (véase 3.2).

4.2.1. Experimentos con valores referencia de los parámetros de *EMP*

En las tablas 4.3 y 4.4 (separadas por motivos de espacio) se pueden ver los resultados de las pruebas experimentales realizadas con el *EMP* sobre los problemas seleccionados, con los distintos valores de los parámetros del esquema metaheurístico que se proponen como valores de referencia para las metaheurísticas *GRASP*, *GA* (Genetic Algorithm), *SS* (Scatter Search), *GRASP + GA*, *GRASP + SS*, *GA + SS* y *GRASP + GA + SS*. Se muestran los resultados medios de 5 ejecuciones del *EMP* para cada conjunto de valores de parámetros.

Los valores mostrados por columnas de cada tabla son, respectivamente, el conjunto de parámetros utilizado y tiempo límite establecido, el problema tratado, el número de iteraciones (véase el pseudocódigo de *EMP* en el cuadro de código 4.1) que le ha dado tiempo a realizar al algoritmo *EMP*, el tiempo medio utilizado por las distintas ejecuciones del algoritmo en milisegundos y el *GAP* medio con respecto a las mejores soluciones encontradas por el algoritmo *GRASP_M* (dos ejecuciones) en un tiempo límite de 100 segundos para los problemas tratados cuyos valores se muestran en la tabla 4.2.

En cuanto a los resultados de las pruebas, empezaremos a comentar los tiempos de ejecución/número de iteraciones del *EMP*. En los problemas de tamaño de problema/solución menores no hay dificultad en cumplir con los tiempos límite de ejecución impuestos (iguales a los tiempos impuestos en los experimentos con *GRASP_M*) pero con el problema de tamaño problema/solución mayor (150×45) todas las metaheurísticas y combinaciones de estas (excepto la metaheurística pura *GA*), según los parámetros

PROBLEMA	MEJOR SOLUCION
GKD-a_50_n15_m12.txt	10145,57839
GKD-a_51_n30_m6.txt	3113,91841
GKD-a_75_n30_m24.txt	16420,41335
SOM-a_20_n50_m15.txt	669
SOM-a_21_n100_m10.txt	331
SOM-a_50_n150_m45.txt	5429

Tabla 4.2: Mejores valores encontrados por GRASP_M para los problemas con los que se ha experimentado con EMP

de referencia dados, no han cumplipo el tiempo límite mínimo pues la fase de inicialización más la primera iteración han superado ampliamente el tiempo límite impuesto.

Analizando los parámetros de cada metaheurística/hibridación se observa como los parámetros de las metaheurísticas que no cumplen con el tiempo máximo establecido incluyen la aplicación de mejoras tanto al conjunto inicial de soluciones (*GRASP*, *SS* e híbridos de estos) como a las mejoras dentro de cada iteración del algoritmo (*SS* e hibridación de este). Además, se puede observar como en todas las metaheurísticas en que interviene *SS*, el número de iteraciones disminuye con respecto a las que no está incluido, de nuevo, debido a la aplicación del proceso de mejora. Todo esto lleva a la conclusión de que el proceso de mejorar las soluciones es el causante de este aumento en el tiempo de obtener resultados. Esto podría indicar que el proceso de mejora pudiera/debería ser optimizado de alguna manera o que no debería ser aplicado en exceso. Si el proceso de mejora no es aplicado en *EMP*, como en el caso de la configuración de parámetros para *GA*, parece suponer la obtención de peores resultados como indican los valores de la tabla 4.3 para la metaheurística pura *GA*. Como información al lector los parámetros de la tabla 4.1 que intervienen en la aplicación del proceso de mejora son: *PEIIni*, *IIEIni* (en la inicialización del problema), *PEIImp* y *IIEImp* (en las distintas iteraciones del algoritmo). Su significado y explicación se provee en [1].

Con respecto a los *GAP's* medios (*fitness*) de las distintas configuraciones de parámetros/metaheurísticas, todo indica que los mejores resultados (GAP más cercano a cero o positivo) se da en aquellas metaheurísticas donde se aplica el proceso de mejora de soluciones, es decir *GRASP* y *SS*, pero a costa, como se a explicado antes, de unos tiempos de ejecución mayores/menor número de iteraciones del algoritmo. Así la metaheurística *GA* es la que peor parada sale en cuanto a calidad de resultados mientras que *SS* y *GRASP + SS* es la que mejor resultados obtiene, en principio, como ya se ha comentado, por su uso más o menos intensivo de la mejora de soluciones. Como cabría esperar, *GRASP* obtiene resultados intermedios (utiliza el proceso de mejora en su fase inicial) mientras que GRASP+GA empeora los resultados de GRASP por sí sóla, por lo menos a corto plazo.

Como conclusión final de este experimento, se podría decir que el factor clave para obtener mejores resultados con *EMP* es la utilización más o menos intensiva del proceso de mejora de soluciones, siempre a costa de aumentar el tiempo necesario para obtener las soluciones. Por tanto, se sugiere la optimización de este proceso de mejora para conseguir un rendimiento eficiente. Por otro lado, es muy interesante la idea de tener un algoritmo donde se puedan aplicar varias metaheurísticas sin la realización de cambios

	PROBLEMA	ITERACIONES	TIEMPO MEDIO (ms.)	GAP
GRASP T = 30	GKD-a_50_n15_m12.txt	226966,8	30000	0,00000000
	GKD-a_51_n30_m6.txt	301881,6	30000	0,00000000
	GKD-a_75_n30_m24.txt	112373,6	30000	0,00000000
	SOM-a_20_n50_m15.txt	180613,8	30000	-0,00029895
	SOM-a_21_n100_m10.txt	241954,4	30000	-0,00060423
	SOM-a_50_n150_m45.txt	1	149907,4	-0,04770676
GRASP T = 100	GKD-a_50_n15_m12.txt	761255,2	100000	0,00000000
	GKD-a_51_n30_m6.txt	1015919,2	100000,6	0,00000000
	GKD-a_75_n30_m24.txt	392475,2	100000	0,00000000
	SOM-a_20_n50_m15.txt	635407,4	100000	0,00000000
	SOM-a_21_n100_m10.txt	825988,6	100000	-0,00181269
	SOM-a_50_n150_m45.txt	1	150455	-0,05017499
GA T = 30	GKD-a_50_n15_m12.txt	16744	30000,6	0,00000000
	GKD-a_51_n30_m6.txt	31259	30000	-0,00592967
	GKD-a_75_n30_m24.txt	7155	30002,4	-0,00022924
	SOM-a_20_n50_m15.txt	13235	30000,8	-0,02242152
	SOM-a_21_n100_m10.txt	19945,6	30000,6	-0,10090634
	SOM-a_50_n150_m45.txt	2874	30003,4	-0,03061337
GA T = 100	GKD-a_50_n15_m12.txt	55921	100000,6	0,00000000
	GKD-a_51_n30_m6.txt	104212,6	100000	-0,00710020
	GKD-a_75_n30_m24.txt	23814,4	100001,8	-0,00007641
	SOM-a_20_n50_m15.txt	43216	100000,6	-0,02571001
	SOM-a_21_n100_m10.txt	66754,6	100000,6	-0,08640483
	SOM-a_50_n150_m45.txt	9671	100005,2	-0,03643397
SS T = 30	GKD-a_50_n15_m12.txt	52,4	30309,4	0,00000000
	GKD-a_51_n30_m6.txt	82,6	30219	0,00000000
	GKD-a_75_n30_m24.txt	5	36526,6	0,00000000
	SOM-a_20_n50_m15.txt	4	33537	0,00000000
	SOM-a_21_n100_m10.txt	6	35541,2	0,00000000
	SOM-a_50_n150_m45.txt	1	753843,4	-0,00814146
SS T = 100	GKD-a_50_n15_m12.txt	173,4	100331,6	0,00000000
	GKD-a_51_n30_m6.txt	274,2	100227	0,00000000
	GKD-a_75_n30_m24.txt	14,2	102018,4	0,00000000
	SOM-a_20_n50_m15.txt	13	104809,4	0,00000000
	SOM-a_21_n100_m10.txt	18	104120,2	0,00000000
	SOM-a_50_n150_m45.txt	1	746458,2	-0,00979923
GRASP + GA T = 30	GKD-a_50_n15_m12.txt	16499,2	30000,4	0,00000000
	GKD-a_51_n30_m6.txt	30990,8	30000	0,00000000
	GKD-a_75_n30_m24.txt	6742,2	30001,8	0,00000000
	SOM-a_20_n50_m15.txt	12397	30000,4	0,00000000
	SOM-a_21_n100_m10.txt	18677,6	30000,2	-0,00181269
	SOM-a_50_n150_m45.txt	1	149478,6	-0,04582796
GRASP + GA T = 100	GKD-a_50_n15_m12.txt	55595,4	100000,8	0,00000000
	GKD-a_51_n30_m6.txt	103287,6	100000	0,00000000
	GKD-a_75_n30_m24.txt	23399,8	100001,2	0,00000000
	SOM-a_20_n50_m15.txt	43354,2	100000,8	0,00000000
	SOM-a_21_n100_m10.txt	65577,6	100000,4	-0,00060423
	SOM-a_50_n150_m45.txt	1	149161,6	-0,04932768

Tabla 4.3: Resultados del primer conjunto de experimentos con EMP

	PROBLEMA	ITERACIONES	TIEMPO MEDIO (ms.)	GAP
GRASP + SS T = 30	GKD-a_50_n15_m12.txt	104	30146,4	0,00000000
	GKD-a_51_n30_m6.txt	164,4	30093,4	0,00000000
	GKD-a_75_n30_m24.txt	8,6	31592,4	0,00000000
	SOM-a_20_n50_m15.txt	7,8	32553,2	0,00000000
	SOM-a_21_n100_m10.txt	10,4	31196,4	0,00000000
	SOM-a_50_n150_m45.txt	1	457619	-0,00884141
GRASP + SS T = 100	GKD-a_50_n15_m12.txt	104	30146,4	0,00000000
	GKD-a_51_n30_m6.txt	164,4	30093,4	0,00000000
	GKD-a_75_n30_m24.txt	8,6	31592,4	0,00000000
	SOM-a_20_n50_m15.txt	7,8	32553,2	0,00000000
	SOM-a_21_n100_m10.txt	10,4	31196,4	0,00000000
	SOM-a_50_n150_m45.txt	1	457619	-0,00884141
GA + SS T = 30	GKD-a_50_n15_m12.txt	32,8	30662,9534	0,00000000
	GKD-a_51_n30_m6.txt	51	30402,6082	0,00000000
	GKD-a_75_n30_m24.txt	3	34060,742	0,00000000
	SOM-a_20_n50_m15.txt	3	38200,7766	0,00000000
	SOM-a_21_n100_m10.txt	4	36896,3708	0,00000000
	SOM-a_50_n150_m45.txt	1	1010870,8432	-0,04052312
GA + SS T = 100	GKD-a_50_n15_m12.txt	107,6	100644,1198	0,00000000
	GKD-a_51_n30_m6.txt	168,6	100227,7258	0,00000000
	GKD-a_75_n30_m24.txt	9	101109,444	0,00000000
	SOM-a_20_n50_m15.txt	8	100962,8326	0,00000000
	SOM-a_21_n100_m10.txt	11	100767,007	0,00000000
	SOM-a_50_n150_m45.txt	1	1017919,2066	-0,04140726
GRASP + GA + SS T = 30	GKD-a_50_n15_m12.txt	32,4	30417,2094	0,00000000
	GKD-a_51_n30_m6.txt	50,8	30304,7252	0,00000000
	GKD-a_75_n30_m24.txt	3	34865,73	0,00000000
	SOM-a_20_n50_m15.txt	3	39326,5742	0,00000000
	SOM-a_21_n100_m10.txt	4	37672,5416	0,00000000
	SOM-a_50_n150_m45.txt	1	1091080,7332	-0,02442439
GRASP + GA + SS T = 100	GKD-a_50_n15_m12.txt	107,6	100548,0322	0,00000000
	GKD-a_51_n30_m6.txt	169,2	100342,5624	0,00000000
	GKD-a_75_n30_m24.txt	9	101189,715	0,00000000
	SOM-a_20_n50_m15.txt	8	101895,9894	0,00000000
	SOM-a_21_n100_m10.txt	11	101373,465	0,00000000
	SOM-a_50_n150_m45.txt	1	1092915,0664	-0,02202984

Tabla 4.4: Resultados del segundo conjunto de experimentos con EMP

al mismo y, si cabe más importante, la adaptación de este algoritmo a distintos tipos de problemas con un mínimo de cambios en el mismo. Por supuesto, puede ser necesario un mayor estudio sobre el tema para afinar el *EMP*.

4.2.2. Análisis de metaheurísticas híbridas

En este apartado realizaremos el mismo tipo de experimentos que realizamos en el apartado anterior pero con unas combinaciones de parámetros distintas a las ofecidas como referencia en [1] y guiadas por la experiencia obtenida con la realización del experimento anterior. Este experimento no intenta ser exhaustivo con respecto al valor de todos los parámetros de *EMP*, sino que se intentará trabajar con respecto a la aplicación del proceso de mejora.

En concreto, se han creado cuatro metaheurísticas híbridas cuyo valor de los parámetros transicionales para *EMP* se muestran en la tabla 4.5.

	Hybrid 1	Hybrid 2	Hybrid 3	Hybrid 4
INEIni	60	60	60	60
FNEIni	30	30	30	30
PEIIni	50	0	0	0
IIEIni	10	0	0	0
MNIEnd	5	5	5	5
NIREnd	2	2	2	2
NBESel	25	25	25	25
NWESel	5	5	5	5
PBBCom	10	10	10	10
PBWCom	10	10	10	10
PWWCom	5	5	5	5
PEIImp	50	50	50	50
IIEImp	20	20	10	15
PDIImp	0	0	0	0
IIDImp	0	0	0	0
NEBInC	15	15	15	15

Tabla 4.5: Valores de los parámetros de metaheurísticas híbridas para *EMP*.

Los parámetros han sido elegidos de acuerdo a las conclusiones obtenidas del experimento anterior. Por tanto, se intenta realizar un uso razonable del proceso de mejora de soluciones teniendo como objetivo reducir el tiempo necesario para la obtención de resultados satisfactorios para el problema *MDP*.

Se ha reducido el tamaño del conjunto inicial de soluciones (*INEIni*) y se ha establecido un tamaño constante para el conjunto de soluciones que se tendrán en cuenta durante las distintas iteraciones del algoritmo (*FNEIni*).

Como ya se dijo anteriormente, los parámetros de la tabla 4.5 que intervienen en la aplicación del proceso de mejora son: *PEIIni*, *IIEIni* (en la inicialización del problema), *PEIImp* y *IIEImp* (en las distintas iteraciones del algoritmo). Se puede observar en la tabla

4.5 que estos parámetros son los que varían en valor entre las distintas metaheurísticas híbridas. Se ha querido dar una mayor preponderancia al proceso de mejora durante las distintas iteraciones del algoritmo frente al realizado en el proceso de inicialización.

Los parámetros relacionados con la combinación de soluciones (los apellidados "*Com*") se han mantenido constantes para reducir variables en este experimento aunque también tienen influencia en el rendimiento del proceso de mejora.

Para una comprensión mejor de los parámetros configurables de *EMP*, una vez más, nos remitimos a [1].

En cuanto al análisis de los resultados de este experimento mostrados en la tabla 4.6, comenzaremos por comparar las metaheurísticas híbridas entre sí y, posteriormente, con las realizadas en el experimento anterior.

Con respecto a los tiempos de ejecución/número de iteraciones realizadas dentro del tiempo límite impuesto podemos observar como la simple eliminación del proceso de mejora en la inicialización del problema en *Hybrid 2* con respecto a *Hybrid 1* supone, en principio, un empeoramiento de los resultados obtenidos, además de no conseguirse un aumento significativo con respecto a los tiempos/número de iteraciones realizadas por el algoritmo. Esto parece indicar que la introducción del proceso de mejora en la inicialización no supone un coste de procesamiento significativo para problemas de tamaño/solución pequeños pero más significativo en problemas más grandes como lo es "SOM-a_50_n150_m45.txt". En cuanto a las metaheurísticas híbridas *Hybrid 3* e *Hybrid 4*, son las que mejor rendimiento dan en cuanto a número de iteraciones/tiempo de ejecución debido al decremento de la intensidad de la mejora de soluciones (*IIEImp*) con respecto a *Hybrid 1* e *Hybrid 2*.

Con respecto a resultados, cabría esperar que la metaheurística *Hybrid 1* fuera la que diera los mejores resultados (por la aplicación del proceso de mejora tanto en la inicialización como en las iteraciones del algoritmo), pero *Hybrid 4* es tanto o más buena que ella en cuanto a *GAP*, siempre y cuando tenga un tiempo límite de ejecución más elevado.

Todo la anterior, parece indicar que la fase de mejora en la inicialización no es tan importante como la fase de mejora de soluciones en las iteraciones del algoritmo *EMP*, que la intensidad de la mejora influye bastante en los tiempos de ejecución del algoritmo y en el "fitness" de las soluciones obtenidas y confirmar, como ya se dijo en el experimento anterior, que el proceso de mejora influye grandemente tanto en los tiempos de ejecución como en la calidad de las soluciones obtenidas para el problema *MDP*, sobre todo en la fase de iteración del algoritmo.

4.3. Conclusión

En este capítulo se ha analizado la aplicación de un esquema metaheurístico parametrizado al problema *MDP*. El esquema parametrizado es versátil en el sentido de que permite trabajar con los mismos conjuntos de parámetros para distintos problemas siempre que se implementen de forma eficiente las funciones básicas del esquema para el problema con el que se trabaje. En los experimentos se ha detectado que la implementación actual no es muy eficiente, por lo que sería necesario optimizar las funciones básicas, quizás utilizando algunas de las ideas del capítulo anterior.

METAHEURISTICA	PROBLEMA	ITERACIONES	TIEMPO	GAP
Hybrid 1 T = 30	GKD-a_50_n15_m12.txt	269,6	30059,6728	0,00000000
	GKD-a_51_n30_m6.txt	425,2	30033,806	0,00000000
	GKD-a_75_n30_m24.txt	22,2	30714,3942	0,00000000
	SOM-a_20_n50_m15.txt	19,6	30550,8038	0,00000000
	SOM-a_21_n100_m10.txt	27	30713,5514	0,00000000
	SOM-a_50_n150_m45.txt	1	168013,7222	-0,00291030
Hybrid 1 T = 100	GKD-a_50_n15_m12.txt	900,4	100055,158	0,00000000
	GKD-a_51_n30_m6.txt	1417,6	100040,9148	0,00000000
	GKD-a_75_n30_m24.txt	73,2	100659,9474	0,00000000
	SOM-a_20_n50_m15.txt	65,4	100853,3202	0,00000000
	SOM-a_21_n100_m10.txt	90,4	100610,6388	-0,00060423
	SOM-a_50_n150_m45.txt	1	152418,6696	-0,00453122
Hybrid 2 T = 30	GKD-a_50_n15_m12.txt	271,4	30069,053	0,00000000
	GKD-a_51_n30_m6.txt	426,2	30044,7978	0,00000000
	GKD-a_75_n30_m24.txt	22,8	31148,5664	0,00000000
	SOM-a_20_n50_m15.txt	19,6	30405,6904	0,00000000
	SOM-a_21_n100_m10.txt	27,6	30650,1726	0,00000000
	SOM-a_50_n150_m45.txt	1	107534,835	-0,01355682
Hybrid 2 T = 100	GKD-a_50_n15_m12.txt	895,8	100066,7064	0,00000000
	GKD-a_51_n30_m6.txt	1420,4	100036,08	0,00000000
	GKD-a_75_n30_m24.txt	73,8	100708,0764	0,00000000
	SOM-a_20_n50_m15.txt	64,8	100601,483	0,00000000
	SOM-a_21_n100_m10.txt	90,6	100454,5704	0,00000000
	SOM-a_50_n150_m45.txt	1	120667,7282	-0,01374102
Hybrid 3 T = 30	GKD-a_50_n15_m12.txt	535	30026,6998	0,00000000
	GKD-a_51_n30_m6.txt	840,6	30019,687	0,00000000
	GKD-a_75_n30_m24.txt	44,6	30414,0274	0,00000000
	SOM-a_20_n50_m15.txt	39,6	30395,1694	0,00000000
	SOM-a_21_n100_m10.txt	54,6	30255,7006	-0,00060423
	SOM-a_50_n150_m45.txt	1	62055,4378	-0,04888561
Hybrid 3 T = 100	GKD-a_50_n15_m12.txt	1778,4	100023,0262	0,00000000
	GKD-a_51_n30_m6.txt	2800,4	100017,818	0,00000000
	GKD-a_75_n30_m24.txt	147	100338,438	0,00000000
	SOM-a_20_n50_m15.txt	131,6	100448,3232	0,00000000
	SOM-a_21_n100_m10.txt	178,8	100249,4314	0,00000000
	SOM-a_50_n150_m45.txt	2	114496,506	-0,01786701
Hybrid 4 T = 30	GKD-a_50_n15_m12.txt	359,8	30045,1962	0,00000000
	GKD-a_51_n30_m6.txt	565,8	30017,5126	0,00000000
	GKD-a_75_n30_m24.txt	29,4	30418,4472	0,00000000
	SOM-a_20_n50_m15.txt	26,6	30443,1636	0,00000000
	SOM-a_21_n100_m10.txt	36,2	30425,8494	0,00000000
	SOM-a_50_n150_m45.txt	1	87557,4048	-0,02818199
Hybrid 4 T = 100	GKD-a_50_n15_m12.txt	1195,8	100033,4668	0,00000000
	GKD-a_51_n30_m6.txt	1888,4	100022,1054	0,00000000
	GKD-a_75_n30_m24.txt	98,4	100660,6192	0,00000000
	SOM-a_20_n50_m15.txt	87,6	100707,4258	0,00000000
	SOM-a_21_n100_m10.txt	120,2	100485,6466	0,00000000
	SOM-a_50_n150_m45.txt	2	171754,1798	-0,00419967

Tabla 4.6: Resultados obtenidos con metaheurísticas híbridas usando EMP

La utilización del esquema parametrizado permite, al igual con el algoritmo **GRASP_M** que hemos implementado, utilizar valores por defecto de los parámetros o experimentar con ellos para determinar valores con los que se obtienen buenos resultados. Los experimentos con el *EMP* se han realizado con tamaños del problema menores que los usados con **GRASP_M**, pero la metodología de trabajo es similar, y permite obtener metaheurísticas híbridas que serán más o menos satisfactorias dependiendo de lo optimizadas que estén las funciones básicas.

Capítulo 5

Conclusiones y trabajos futuros

En este trabajo se ha introducido, brevemente, al problema *MDP* (*Maximun Diversity Problem*) y se ha visto su importancia en cuanto a la resolución de situaciones de la vida cotidiana. El problema *MDP* pertenece a un grupo de problemas de optimización combinatoria que, generalmente, no son abordables por medio de algoritmos deterministas debido a su complejidad. Es por ello, que se recurre frecuentemente a la utilización de heurísticas y metaheurísticas para su resolución eficiente, tanto en relación al tiempo de ejecución como en la calidad de los resultados obtenidos.

Por medio de este trabajo se ha querido desarrollar una heurística/metaheurística especialmente diseñada para el abordamiento del problema *MDP*, así como realizar un estudio de otro algoritmo con un enfoque diferente a lo normalmente utilizado en este campo de estudio. Este algoritmo es un "*Esquema Metaheurístico Parametrizado*" (*EMP*) que como su propio nombre indica se basa en unos parámetros llamados "parámetros transicionales". Gracias a la asignación de valores a estos parámetros y al nivel de abstracción del algoritmo, se permite la implementación de varias metaheurísticas, hibridaciones entre ellas e, incluso, la posibilidad de encontrar nuevas metaheurísticas aplicables a varios tipos de problemas de optimización combinatoria sin apenas tener que realizar cambios en el código del algoritmo.

En el capítulo 3 se ha desarrollado y analizado un nuevo algoritmo al que se le ha dado el nombre "GRASP_M". Se ha intentado abordar el problema desde un punto de vista original, sin apenas tener en cuenta, a priori, nada de lo hecho anteriormente. Durante su desarrollo se ha realizado una experimentación continuada con la prueba de varias ideas surgidas y un gran énfasis en la optimización de los procesos intervinientes en el algoritmo. Se cree que se ha conseguido realizar un buen algoritmo de resolución del problema *MDP* a la vista de los resultados obtenidos en las pruebas realizadas (véase el apartado 3.2) y teniendo en cuenta el escaso tiempo límite de ejecución del algoritmo.

En base a lo anterior, se propone un poco más de iniciativa/creatividad en la búsqueda de algoritmos capaces de resolver problemas de tipo combinatorio como lo es el problema *MDP*. No se intenta decir que el algoritmo creado es el mejor, pero a la vista de los buenos resultados obtenidos con nuestro algoritmo se cree que a partir de nuevas ideas se pueden conseguir buenos resultados en la resolución de este tipo de problemas y no sólo aplicar explícitamente las metaheurísticas comunes o una hibridación directa de estas. Esta es la principal conclusión del intento de creación de un algoritmo para la resolución

del problema *MDP*, uno de los tantos tipos de problema de optimización combinatoria. Si se ha conseguido aportar algo en el proceso de creación de este algoritmo se da por satisfecha la realización de este proyecto.

Como posibles trabajos futuros en cuanto al algoritmo desarrollado, se propone la investigación de nuevas formas de realizar los procesos, principalmente el proceso de mejora, que es el que mayor tiempo de procesamiento consume. Quizás se haya aplicado un excesivo uso de la aleatoriedad en este algoritmo. Podría ser que la aleatoriedad sea, intrínsicamente, necesaria para este tipo de problemas pero se cree, tras la realización del algoritmo, que, en ciertas ocasiones, esta aleatoriedad podría ser sustituida por algún tipo de "inteligencia conductora" que la mejorara. Tampoco se ha tenido en cuenta lo mejor de las distintas metaheurísticas, principalmente, aplicadas al problema *MDP*. Podría ser que algunos aspectos de estas pudieran mejorar el algoritmo aportado.

También se han realizado pruebas sobre un "Esquema Metaheurístico parametrizado"(*EMP*) [1] que intenta proveer una solución al uso de distintas metaheurísticas con diversos tipos de problemas de optimización combinatoria con escasos cambios en el algoritmo inicial en base a una serie de parámetros. Estas pruebas, en general, han dado como resultado una menor eficiencia con respecto al nuevo algoritmo presentado pero la idea propuesta es bastante buena. Es necesaria una mayor experimentación en base a los valores de los parámetros del algoritmo, a la vez que una mayor optimización de las funciones utilizadas en la resolución del *MDP*. Esto podría dar una mayor eficiencia en la resolución de nuestro problema, al mismo tiempo que contribuiría a contrastar la validez del *EMP*.

Tanto para el algoritmo *GRASP_M* como para el esquema *EMP*, se han realizado experimentos para determinar valores de los parámetros satisfactorios. Los experimentos se han realizado de manera "manual", es decir, realizando experimentos con un conjunto de parámetros, obteniendo conclusiones, y realizando experimentos con otro conjunto de parámetros, y así repetidamente. Esto conlleva un alto tiempo de experimentación supervisado por el usuario. Una posibilidad puede ser desarrollar una hiperheurística para la obtención automática de valores satisfactorios de los parámetros. Esto se ha realizado para el *EMP* en [7], y la misma idea podría aplicarse a *GRASP_M*.

Hemos visto que el aumento de tiempo de ejecución permite realizar más evaluaciones y mejoras y por lo tanto contribuye a obtener mejores soluciones. El uso de algoritmos paralelos permitiría reducir el tiempo de ejecución o realizar más evaluaciones en el mismo tiempo. Esquemas paralelos para memoria compartida y paso de mensajes se han desarrollado para el *EMP* [5, 6], lo que permite optimizar de forma unificada la paralelización de varias metaheurísticas. Las mismas ideas se podrían aplicar para la paralelización y optimización del algoritmo *GRASP_M*, lo que permitiría mejorar aun más los resultados obtenidos con él.

Bibliografía

[1] ALMEIDA, F., GIMÉNEZ, D., LÓPEZ-ESPÍN, J. J., AND PÉREZ-PÉREZ, M. Parameterized schemes of metaheuristics: Basic ideas and applications with Genetic Algorithms, Scatter Search, and GRASP. *Systems, Man, and Cybernetics: Systems, IEEE Transactions on 43*, 3 (2013), 570–586.

[2] ARINGHIERI, R., CORDONE, R., AND MELZANI, Y. Tabu Search versus GRASP for the maximum diversity problem. *4OR 6*, 1 (2008), 45–60.

[3] BAGHEL, M., AGRAWAL, S., AND SILAKARI, S. Article: Survey of metaheuristic algorithms for combinatorial optimization. *International Journal of Computer Applications 58*, 19 (November 2012), 21–31.

[4] CUTILLAS-LOZANO, J.-M., FRANCO, M.-A., AND GIMÉNEZ, D. Comparing variable width backtracking and metaheuristics, experiments with the maximum diversity problem. In *GECCO Companion* (2015), ACM. Late Breaking Abstract.

[5] CUTILLAS-LOZANO, J. M., AND GIMÉNEZ, D. Optimizing shared-memory hyperheuristics on top of parameterized metaheuristics. In *Proceedings of the International Conference on Computational Science, ICCS 2014, Cairns, Queensland, Australia, 10-12 June, 2014* (2014), pp. 20–29.

[6] CUTILLAS-LOZANO, J. M., AND GIMÉNEZ, D. Parameterized message-passing metaheuristic schemes on a heterogeneous computing system. In *Theory and Practice of Natural Computing - Third International Conference, TPNC 2014, Granada, Spain, December 9-11, 2014. Proceedings* (2014), pp. 59–70.

[7] CUTILLAS-LOZANO, J. M., GIMÉNEZ, D., AND ALMEIDA, F. Hyperheuristics based on parametrized metaheuristic schemes. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015* (2015), pp. 361–368.

[8] DUARTE, A., AND MARTÍ, R. Tabu search and GRASP for the maximum diversity problem. *European Journal of Operational Research 178*, 1 (2007), 71–84.

[9] DÍAZ-FERNÁNDEZ, A. *Optimización heurística y redes neuronales*. Ediciones Paraninfo S.A., 1996.

[10] ERCUT, E., AND NEUMAN, S. Analytic models for locating undesirable facilities. *European Journal of Operational Research 40* (1989), 275–291.

[11] ERKUT, E. The discrete p-dispersion problem. *European Journal of Operational Research 46*, 1 (1990), 48 – 60.

[12] FEO, T. A., AND RESENDE, M. G. C. A probabilistic heuristic for a computationally difficult set covering problem. *Oper. Res. Lett. 8*, 2 (Apr. 1989), 67–71.

[13] GALLEGO, M., DUARTE, A., LAGUNA, M., AND MARTÍ, R. Hybrid heuristics for the maximum diversity problem. *Computational Optimization and Applications 44*, 3 (2009), 411–426.

[14] GHOSH, J. B. Computational aspects of the maximum diversity problem. *Operations Research Letters 19*, 4 (1996), 175 – 181.

[15] GLOVER, F. Heuristics for integer programming using surrogate constraints. *Decision Sciences 8*, 1 (1977), 156–166.

[16] GLOVER, F. Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research 13*, 5 (1986), 533 – 549.

[17] GLOVER, F. Tabu search: a tutorial. *Interfaces*, 20 (1990), 74 – 94.

[18] GLOVER, F. Tabu search for nonlinear and parametric optimization (with links to genetic algorithms). *Discrete Applied Mathematics 49*, 1 (1994), 231–255.

[19] GLOVER, F. A template for scatter search and path relinking. In *Artificial Evolution*, J.-K. Hao, E. Lutton, E. Ronald, M. Schoenauer, and D. Snyers, Eds., vol. 1363 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1998, pp. 1–51.

[20] GLOVER, F., KUO, C. C., AND DHIR, K. Heuristic Algorithms for the Maximum Diversity Problem. *Journal of Information and Optimization Sciences 19(1)* (1998), 109–132.

[21] GRANVILLE, V., KRIVANEK, M., AND RASSON, J.-P. Simulated annealing: a proof of convergence. *Pattern Analysis and Machine Intelligence, IEEE Transactions on 16*, 6 (Jun 1994), 652–656.

[22] HANSEN, P., MLADENOVIC, N., AND MORENO PÉREZ, J. A. Búsqueda de entorno variable. *Inteligencia Artificial. Revista Iberoamericana de Inteligencia Artificial*, 19 (2003), 77 – 92.

[23] HONG, L., AND PAGE, S. E. Groups of diverse problem solvers can outperform groups of high-ability problem solvers. *Proceedings of the National Academy of Sciences of the United States of America 101*, 46 (2004), 16385–16389.

[24] KINCAID, R. Good solutions to discrete noxious location problems via metaheuristics. *Annals of Operations Research 40*, 1 (1992), 265–281.

[25] KIRKPATRICK, S., GELATT, C. D., AND VECCHI, M. P. Optimization by simulated annealing. *Science 220*, 4598 (1983), 671–680.

[26] KORST, J. H. M., AARTS, E. H. L., AND MICHIELS, W. Simulated annealing. In *Handbook of Approximation Algorithms and Metaheuristics.*, T. F. Gonzalez, Ed. Chapman and Hall/CRC, 2007.

[27] KUO, C. C., GLOVER, F., AND DHIR, K. S. Analyzing and modeling the maximum diversity problem by zero-one programming. *Decision Sciences 24*, 6 (1993), 1171–1185.

[28] LEE, T. Q., AND PARK, Y. T. A similarity measure for collaborative filtering with implicit feedback. *ICIC 2007* (2007), 385–397.

[29] LOZANO, M., MOLINA, D., AND GARCÍA-MARTÍNEZ, C. Iterated greedy for the maximum diversity problem. *European Journal of Operational Research 214*, 1 (2011), 31–38.

[30] MARTÍ, R., GALLEGO, M., DUARTE, A., AND PARDO, E. G. Heuristics and metaheuristics for the maximum diversity problem. *Journal of Heuristics 19*, 4 (2013), 591–615.

[31] MARTÍ, R., LAGUNA, M., AND GLOVER, F. Principles of scatter search. *European Journal of Operational Research 169*, 2 (2006), 359–372.

[32] MARTÍ, R. Multi-start methods. In *Handbook of Metaheuristics*, F. Glover and G. Kochenberger, Eds., vol. 57 of *International Series in Operations Research & Management Science*. Springer US, 2003, pp. 355–368.

[33] MARTÍ, R., GALLEGO, M., AND DUARTE, A. Optsicom project. maximum diversity problem reference set. `http://www.optsicom.es/mdp/mdplib_2010.zip`. Último acceso: Mayo 2015.

[34] MARTÍ, R., GALLEGO, M., AND DUARTE, A. A branch and bound algorithm for the maximum diversity problem. *European Journal of Operational Research 200*, 1 (2010), 36 – 44.

[35] MEINL, T. *Maximum-score diversity selection*. PhD thesis, University of Konstanz, 2010. `http://kops.ub.uni-konstanz.de/volltexte/2010/12211/`. Último acceso: Mayo 2015.

[36] MLADENOVIĆ, N., AND HANSEN, P. Variable neighborhood search. *Computers & Operations Research 24*, 11 (1997), 1097 – 1100.

[37] PALUBECKIS, G. Iterated tabu search for the maximum diversity problem. *Applied Mathematics and Computation 189*, 1 (2007), 371–383.

[38] PUCHINGER, J., AND RAIDL, G. Combining metaheuristics and exact algorithms in combinatorial optimization: A survey and classification. In *Artificial Intelligence and Knowledge Engineering Applications: A Bioinspired Approach*, J. Mira and J. Álvarez, Eds., vol. 3562 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2005, pp. 41–53.

[39] SANDOYA, F. *Un modelo para el problema de la diversidad máxima. Un estudio para la selección de lo mejor y lo más diverso*. PhD thesis, Ingeniería de Sistemas - Investigación de Operaciones. Universidad Nacional Autónoma de Mexico, Dec. 2013.

[40] SANTOS, L., RIBEIRO, M., PLASTINO, A., AND MARTINS, S. A Hybrid GRASP with Data Mining for the Maximum Diversity Problem. In *Hybrid Metaheuristics*, M. J. Blesa, C. Blum, A. Roli, and M. Sampels, Eds., vol. 3636 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2005, pp. 116–127.

[41] SILVA, G., DE ANDRADE, M., OCHI, L., MARTINS, S., AND PLASTINO, A. New heuristics for the maximum diversity problem. *Journal of Heuristics 13*, 4 (2007), 315–336.

[42] WANG, Y., HAO, J.-K., GLOVER, F., AND LÜ, Z. A tabu search based memetic algorithm for the maximum diversity problem. *Engineering Applications of Artificial Intelligence 27*, 0 (2014), 103 – 114.

Apéndice A

Manual de usuario para el algoritmo GRASP_M

Además de las funciones principales y estructuras de datos necesarias para el funcionamiento y utilización del algoritmo *GRASP_M* se ha creado un pequeño programa para facilitar su uso en la realización de las pruebas realizadas en este trabajo. A continuación se mostrará el manual de usuario para quien quiera o pueda estar interesado en la utilización de este pequeño programa.

El programa generado (se provee archivo "Makefile") tiene el nombre "**MDP**" y toma una serie de argumentos para facilitar el trabajo con la resolución del problema *MDP*. Estos argumentos y su significado son los siguientes:

- **-f** *ruta/nombreArchivo*. Permite especificar un archivo de distancias determinado a tratar.
- **-l** *ruta/nombreArchivoLotes*. Permite especificar un archivo donde se encontrarán una o varias rutas y nombres de archivo de distancias a tratar. De esta forma se permite la resolución de varios problemas MDP en una sola ejecución del programa.
- **-c** *ruta/nombreArchivoConfiguracion*. Permite especificar un archivo que contendrá valores para alterar la configuración por defecto del algoritmo.
- **-s** *ruta/nombreArchivoSalida*. Permite especificar un archivo donde se almacenarán los resultados de la ejecución del algoritmo en el formato especificado en 3.1.1. Si no se especifica un archivo de salida, el resultado de la ejecución del algoritmo será mostrado por pantalla.
- **-a** $a|A|s|S$. Permite especificar el modo de escritura del archivo de salida. Si se especifica el modo '*a*' o '*A*', el nuevo contenido generado se añadirá a lo ya existente en el fichero de salida pero si se especifica '*s*' o '*S*', el nuevo contenido sobreescribirá a lo ya existente en el fichero si lo hubiera. Si no se especificara nada, el comportamiento por defecto es el de añadir contenido al fichero.
- **-r**. Especifica que se trunque el fichero de salida, si se hubiera indicado uno con el parámetro **-s**, antes de iniciar la ejecución del algoritmo, es decir, comenzar con un fichero de salida limpio, independientemente de lo especificado con el parámetro **-a**.

- **-t** *numSegundos*. Permite especificar el tiempo límite en segundos para cada problema tratado por el algoritmo en esta ejecución. El tiempo por defecto es de 200 segundos. Este tiempo también puede ser especificado a través del fichero de configuración especificado con **-c**. En este caso el tiempo especificado a través del fichero de configuración tiene precedencia sobre el parámetro **-t**.

- **-n** *numRepeticiones*. Permite especificar el número de repeticiones al tratar cada archivo de distancias especificado, es decir, cada problema será resuelto *numRepeticiones* consecutivas antes de pasar a resolver el siguiente problema especificado, si lo hubiera.

Casi todos los parámetros esperan tener un argumento a continuación, excepto **-r**. Si no se proveen estos parámetros se puede producir un error o un comportamiento inesperado del programa. Los especificadores de parámetro no son sensibles a mayúsculas/-minúsculas, por tanto deben ser escritos tal cual se especifican en estas instrucciones.

En los parámetros donde se especifica una *ruta+nombre de archivo* se establecido un límite de 499 caracteres en la longitud máxima de la cadena de texto especificada. Siempre podría ser modificable este límite modificando el código fuente del programa.

Una ejecución típica con un sólo archivo de distancias, con los parámetros por defecto provistos por el algoritmo, con la salida a un fichero de texto, truncando este fichero, por si ya existiera uno anteriormente con el mismo nombre y no estuviera vacío, y no se deseara conservar y con un tiempo límite de 100 segundos sería:

./MDP -f archivoDistancias -s salida.csv -r -t 100

Si ahora se quisiera añadir a ese archivo de salida la ejecución de otro problema con la especificación de otra configuración del algoritmo y que este problema fuera tratado 5 veces, la invocación al programa sería la siguiente:

./MDP -f archivoDistancias1 -c archivoConfiguracion -s salida.csv -a a -n 5

Si se quisiera tratar un conjunto de problemas en una sola llamada al programa con salida a un archivo de texto y todo lo demás por defecto:

./MDP -l archivoLotes -s salida.csv

A.1. Formato archivo por lotes

La especificación del archivo por lotes a pasar al parámetro **-l** es la de un simple archivo de texto con una ruta/nombre de archivo por cada línea del archivo. No se permiten otro tipo de contenido, ni comentarios ni nada más. El contenido de este archivo podría ser, por ejemplo, el especificado en el cuadro de código A.1.

```
../MDG-c/MDG-c_1_n3000_m300.txt
../MDG-c/MDG-c_2_n3000_m300.txt
../MDG-c/MDG-c_3_n3000_m300.txt
../MDG-c/MDG-c_4_n3000_m300.txt
../MDG-c/MDG-c_5_n3000_m300.txt
../MDG-c/MDG-c_6_n3000_m400.txt
../MDG-c/MDG-c_7_n3000_m400.txt
../MDG-c/MDG-c_8_n3000_m400.txt
../MDG-c/MDG-c_9_n3000_m400.txt
../MDG-c/MDG-c_10_n3000_m400.txt
../MDG-c/MDG-c_11_n3000_m500.txt
../MDG-c/MDG-c_12_n3000_m500.txt
../MDG-c/MDG-c_13_n3000_m500.txt
../MDG-c/MDG-c_14_n3000_m500.txt
../MDG-c/MDG-c_15_n3000_m500.txt
../MDG-c/MDG-c_16_n3000_m600.txt
../MDG-c/MDG-c_17_n3000_m600.txt
../MDG-c/MDG-c_18_n3000_m600.txt
../MDG-c/MDG-c_19_n3000_m600.txt
../MDG-c/MDG-c_20_n3000_m600.txt
```

Código A.1: Ejemplo de archivo por lotes

A.2. Formato archivo de configuración

Como hemos visto anteriormente, se puede especificar que el algoritmo se ejecute con una configuración distinta a la que posee por defecto mediante el parámetro **-c**. El formato del archivo pasado a este parámetro es un simple archivo de texto donde se pueden incluir comentarios comenzando la línea con el símbolo **#**. Los distintos aspectos del algoritmo a configurar se especifican mediante el nombre del parámetro a configurar y en la siguiente línea el valor del parámetro. Si no se especifica el nombre de un parámetro o se comenta no se cambiará su configuración. Tanto los nombres de los parámetros como la asignación de los valores a cada uno de ellos y una breve explicación de su significado se muestran en un fichero de configuración de ejemplo en el cuadro de código A.2.

```
1  #ARCHIVO DE CONFIGURACION POR DEFECTO PARA EL PROBLEMA MDP.
2
3  #Numero a guardar en conjunto de soluciones del problema.
4  MDP_NSC
5  5
6
7  #Valor FLAGS para el problema (unsigned int).
8  #Flags por defecto (MDP_FSDO | MDP_FSDOD | MDP_FDOO | MDP_FDOOD | MDP_FEPS)
9  MDP_FLAGS
10 179
11
12 #Numero de iteraciones sin mejora antes de aplicar el reinicio de solucion.
13 MDP_MISM
14 50
15
16 #Porcentaje (tanto por 1) de mejoras con respecto a numero de soluciones iniciales.
17 MDP_PM
18 0.10
19
20 #Tiempo maximo (en segundos) para algoritmo MDP (sin fase de preparacion).
21 MDP_TMAX
22 100¿
23
24 #Aplicar PR a conjunto de soluciones final? (No cuenta como tiempo de algoritmo).
       Valores validos (true, false) en minusculas.
25 MDP_PRF
26 false
27
28 #Digitos de precision en comparacion de valores de solucion. Valido de 1 a 10. (0
       significa una diferencia menor a 1 para ser considerados iguales)
29 MDP_REC
30 6
```

Código A.2: Ejemplo de archivo de configuración

A.3. El archivo de salida

Ya se especificó el formato de la salida de los resultados a un archivo de texto separado por comas en la sección 3.1.1 de este trabajo. Aquí solamente comentaremos que cada resultado de ejecución del algoritmo se guardará en una línea del fichero y como ejemplo de esta salida mostraremos una fichero de salida de ejemplo en el cuadro de código A.3.

```
1 MDG-c_1_n3000_m300.txt, 3000, 300, 300, 304486, 300493, 105760, 24892586.000000, 14, 21,
     23, 39, 48, 58, 67, 71, 72, 79, 84, 102, 103, 108, 126, 132, 145, 148, 150, 155,
    161, 162, 164, 171, 172, 178, 191, 205, 211, 222, 256, 261, 263, 265, 280, 283, 284,
     321, 328, 333, 337, 350, 356, 359, 370, 382, 393, 402, 415, 419, 436, 449, 451,
    455, 467, 480, 487, 488, 494, 495, 510, 514, 519, 532, 535, 539, 546, 547, 564, 574,
     582, 588, 597, 611, 642, 664, 682, 694, 699, 711, 714, 715, 717, 745, 754, 762,
    765, 775, 776, 789, 791, 809, 815, 840, 856, 872, 885, 888, 892, 903, 911, 920, 946,
     947, 961, 968, 972, 996, 1004, 1013, 1014, 1015, 1031, 1033, 1034, 1038, 1067,
    1068, 1069, 1079, 1094, 1097, 1101, 1114, 1138, 1141, 1147, 1149, 1155, 1158, 1159,
    1166, 1169, 1170, 1172, 1173, 1184, 1185, 1207, 1217, 1258, 1263, 1274, 1301, 1306,
    1324, 1329, 1336, 1353, 1379, 1381, 1382, 1396, 1398, 1401, 1407, 1411, 1412, 1413,
    1415, 1438, 1445, 1462, 1466, 1469, 1528, 1535, 1548, 1549, 1555, 1556, 1558, 1574,
    1577, 1578, 1591, 1595, 1613, 1617, 1620, 1622, 1634, 1638, 1645, 1648, 1656, 1661,
    1663, 1671, 1695, 1739, 1742, 1744, 1755, 1756, 1770, 1771, 1774, 1783, 1797, 1814,
    1815, 1829, 1873, 1882, 1887, 1937, 1974, 1988, 1998, 2005, 2007, 2017, 2025, 2036,
    2044, 2047, 2056, 2071, 2087, 2109, 2152, 2178, 2181, 2192, 2203, 2220, 2247, 2254,
    2255, 2283, 2288, 2292, 2307, 2312, 2321, 2323, 2324, 2356, 2357, 2358, 2370, 2386,
    2413, 2453, 2454, 2501, 2502, 2514, 2515, 2517, 2522, 2524, 2526, 2533, 2548, 2561,
    2573, 2579, 2599, 2602, 2606, 2607, 2627, 2640, 2644, 2645, 2662, 2663, 2673, 2676,
    2679, 2683, 2686, 2708, 2714, 2722, 2736, 2747, 2764, 2773, 2779, 2804, 2808, 2809,
    2815, 2817, 2834, 2836, 2837, 2842, 2844, 2849, 2852, 2871, 2933, 2934, 2937, 2952,
    2980
2 MDG-c_2_n3000_m300.txt, 3000, 300, 300, 304310, 300034, 54961, 24879845.000000, 1, 4,
    11, 21, 30, 35, 40, 45, 52, 60, 67, 68, 69, 94, 95, 97, 98, 106, 115, 121, 125, 142,
     154, 160, 167, 173, 182, 185, 194, 203, 210, 219, 220, 232, 250, 253, 261, 267,
    270, 287, 314, 324, 325, 326, 329, 331, 335, 348, 349, 351, 362, 366, 372, 373, 376,
     395, 402, 403, 407, 429, 437, 451, 453, 473, 479, 488, 499, 501, 521, 546, 548,
    564, 578, 589, 595, 603, 622, 628, 643, 661, 665, 690, 698, 700, 723, 748, 772, 786,
     787, 795, 797, 813, 819, 823, 837, 843, 849, 850, 851, 854, 860, 861, 865, 866,
    874, 915, 924, 940, 945, 951, 969, 971, 979, 989, 993, 995, 1042, 1066, 1074, 1101,
    1108, 1131, 1133, 1136, 1138, 1145, 1150, 1151, 1158, 1173, 1181, 1186, 1197, 1198,
    1207, 1219, 1242, 1256, 1258, 1297, 1323, 1324, 1330, 1338, 1346, 1350, 1351, 1363,
    1364, 1366, 1371, 1373, 1378, 1383, 1396, 1397, 1438, 1453, 1455, 1483, 1508, 1513,
    1518, 1539, 1561, 1567, 1569, 1577, 1583, 1584, 1607, 1615, 1629, 1636, 1640, 1648,
    1677, 1693, 1711, 1731, 1736, 1740, 1742, 1743, 1783, 1785, 1803, 1818, 1819, 1823,
    1833, 1835, 1856, 1862, 1865, 1869, 1870, 1884, 1885, 1889, 1890, 1901, 1911, 1926,
    1939, 1945, 1953, 1965, 1978, 1982, 1987, 1991, 2018, 2030, 2046, 2055, 2067, 2068,
    2084, 2097, 2106, 2109, 2116, 2128, 2149, 2184, 2188, 2194, 2198, 2202, 2208, 2220,
    2274, 2301, 2318, 2324, 2332, 2352, 2362, 2375, 2380, 2382, 2391, 2392, 2400, 2407,
    2409, 2414, 2432, 2455, 2478, 2485, 2495, 2497, 2498, 2503, 2524, 2533, 2534, 2546,
    2553, 2554, 2556, 2574, 2590, 2591, 2605, 2615, 2624, 2634, 2636, 2665, 2690, 2698,
    2707, 2740, 2751, 2768, 2802, 2805, 2806, 2817, 2823, 2847, 2850, 2855, 2878, 2887,
    2904, 2913, 2917, 2919, 2923, 2926, 2943, 2946, 2953, 2957, 2969, 2982
```

Código A.3: Ejemplo de fichero de salida

Apéndice B

Manual de usuario para el algoritmo EMP

El programa utilizado para las pruebas con el algoritmo *EMP* es bastante más sencillo de utilizar que el comentado en el apéndice anterior para el algoritmo GRASP_M.

Tan sólo admite tres argumentos como parámetros para su ejecución. El nombre del fichero ejecutado (se incluye archivo *Makefile*) es "*metaplotresul*". Los parámetros son, en estricto orden de entrada por línea de comandos:

1. La ruta/nombre del fichero de distancias perteneciente al problema a tratar por el algoritmo *EMP*.
2. La ruta/nombre del fichero de configuración donde se especifican los distintos valores de los "*parámetros transicionales*" utilizados por *EMP* y que se especifican en la tabla 4.1 de este documento.
3. La ruta/nombre del fichero de salida con valores separados por comas donde se almacenarán los resultados de las ejecuciones del algoritmo. Cada resultado de salida se irá añadiendo al archivo especificado, es decir, no se sobreescribirá el contenido que pudiera ya tener este archivo.

El formato del archivo de distancias es el mismo utilizado para *GRASP_M* y es el provisto por [33].

En el cuadro de código B.1 se muestran varios ejemplos de llamada al programa para ejecutar el algoritmo EMP con distintos tipos de poblemas *MDP*, con distintos archivos de configuración y distintos archivos de salida.

```
1 ./metaplotresul ./GKD-a/GKD-a_75_n30_m24.txt ga_ss_30 ga_ss_30.csv
2 ./metaplotresul ./SOM-a/SOM-a_21_n100_m10.txt grasp_ss_100 grasp_ss_100.csv
3 ./metaplotresul ./SOM-a/SOM-a_50_n150_m45.txt Hybrid3 Hybrid3.csv
```

Código B.1: Ejemplo de llamadas de ejecución el algoritmo EMP

B.1. Formato del archivo de configuración

El archivo de configuración es un simple archivo de texto que consta de 22 valores.

El primer valor es un valor que no interviene en la configuración sino que es un número que se le asigna a la metaheurística en concreto especificada mediante la configuración del algoritmo. No tiene mayor importancia ni efecto en la ejecución del algoritmo, tan sólo que si su valor es igual a -1 no se llevará a cabo la ejecución del programa.

Los siguientes 16 valores son los valores de los parámetros de configuración del algoritmo en el mismo orden que se muestran en la tabla 4.1:

1. INEIni
2. FNEIni
3. PEIIni
4. IIEIni
5. MNIEnd
6. NIREnd
7. NBESel
8. NWESel
9. PBBCom
10. PBWCom
11. PWWCom
12. PEIImp
13. IIEImp
14. PDIImp
15. IIDImp
16. NEBInC

Los siguientes 4 valores son siempre establecidos a 0 en este trabajo porque pertenecen a parámetros de configuración de *EMP* no documentados y que no han sido tenidos en cuenta.

El último parámetro es el valor del tiempo límite, expresado en segundos, establecido para la resolución del problema *MDP* por parte del algoritmo *EMP*.

Se muestra un fichero de configuración de ejemplo para la ejecución de una metaheurística *GRASP*, según los valores de referencia especificados para ella por [1], en el cuadro de código B.2.

```
1
100
10
100
10
100
50
10
0
5
0
0
0
0
0
0
10
0
0
0
0
30
```

Código B.2: Ejemplo de archivo de configuración para EMP con metaheurística GRASP

B.2. El archivo de salida

El archivo de salida es un simple archivo de texto con el formato de valores separados por comas. Cada solución a un problema *MDP* realizado por el algoritmo *EMP* se guarda en una línea distinta del fichero. El significado de los campos y su orden difieren un poco con respecto a la salida del algoritmo *GRASP_M* y se explican en la siguiente lista (orden del campo entre paréntesis):

(1). Nombre del fichero de distancias perteneciente al problema tal y como se introduce en la línea de comandos como primer argumento del programa.

(2). Nombre del fichero de configuración de los parámetros del algoritmo *EMP* tal y como se introduce como segundo argumento del programa.

(3). Tamaño del problema n.

(4). Tamaño de la solución al problema m.

(5). Número de iteraciones realizadas por el algoritmo *EMP*.

(6). Número de iteraciones con repetición consecutiva de soluciones realizadas por el algoritmo *EMP* a su finalización.

(7). Tiempo límite en milisegundos establecido para el algoritmo *EMP* en esta ejecución.

(8). Tiempo real en milisegundos consumido por el algoritmo *EMP* en esta ejecución.

(9). Valor de la solución encontrada para esta ejecución del algoritmo *EMP*.

(10..10+m). Elementos de la solución encontrada enumerados en orden ascendente.

En el cuadro de código B.3 se muestra un ejemplo de salida de dos ejecuciones del algoritmo *EMP* sobre el mismo fichero de salida.

```
1 ./GKD-a/GKD-a_50_n15_m12.txt,ga_30,15,12,16732,16730,30000,30001,10145.57839,
      0,1,2,3,4,5,6,8,11,12,13,14
2 GKD-a/GKD-a_75_n30_m24.txt,ga_30,30,24,7167,7156,30000,30000,16414.13977999999,
      0,2,3,4,5,7,8,10,11,12,13,14,15,16,17,18,19,21,22,23,24,25,26,28
```

Código B.3: Ejemplo de archivo de salida de EMP