Introducción a la Programación

Tema 16. Algoritmos de Aproximación

1. Introducción

En los capítulos anteriores hemos visto distintos tipos de algoritmos recursivos: *Divide y Vencerás, Programación Dinámica* y algoritmos de *Vuelta Atrás* con sus diferentes variantes. Son algoritmos que usan una búsqueda exhaustiva. Para el caso de problemas de optimización son capaces de encontrar, si existe, la solución exacta. Pero en algunos casos la complejidad del algoritmo es demasiado alta y debemos buscar otro tipo de algoritmos que encuentren soluciones subóptimas cercanas a la óptima pero con un menor coste de ejecución. Llamaremos a este tipo *Algoritmos de Aproximación*. En este tema veremos *Algoritmos Iterativos de Aproximación* que serán de dos tipos: *Simulated Annealing y Algoritmos Genéticos.* Los dos son algoritmos iterativos. Junto a ellos podemos considerar los algoritmos *Voraces* que también son en general algoritmos iterativos de aproximación pero que vemos en otro capítulo.

En este capítulo veremos también la *Programación Lineal Entera* que es una técnica para encontrar soluciones exactas o aproximadas relajando algunas de las restricciones.

Los *algoritmos de aproximación* son adecuados para resolver problemas de optimización. Este tipo de problemas viene definido por un conjunto de restricciones y una función objetivo que calcula un valor para cada una de las posibles soluciones. En un problema de optimización buscamos la solución, que cumpliendo las restricciones del problema, minimice (o maximice) el valor de la función objetivo.

En general, como iremos viendo con ejemplos, los problemas que no son de optimización pero buscan una o varias soluciones que cumplan las restricciones del problema (o solamente la existencia de alguna de ellas) pueden transformarse en problemas de optimización.

En los algoritmos de aproximación diseñamos un estado cuyos valores representarán las posibles soluciones del problema. La función objetivo se evaluará sobre cada una de las instancias del estado diseñado. Existirá, además, una función que aplicada a cada estado nos indique si representa una solución válida o no. Si el estado representa una solución válida diremos que es un estado válido. En otro caso es un estado inválido. Dispondremos también otra función que calculará la solución asociada a los estados válidos.

Los algoritmos de aproximación que veremos son iterativos. Usan una *condición de parada* para decidir cuando terminan.

Las estrategias que estudiaremos son: *Simulated Annealing y Algoritmos Genéticos.* En la primera se diseñan un conjunto de alternativas posibles, también podemos llamarlas operadores, para cada estado, se escoge aleatoriamente una alternativa de entre las posibles, y se pasa al estado siguiente. Si se acepta se continua. Si no se acepta se vuelve al estado anterior. Es necesario definir una *condición de aceptación* del estado siguiente. Los *Algoritmos Genéticos* emulan los mecanismos de la evolución. Codifican los estados posibles en distintas formas y usan, como en la evolución operadores (similares a las alternativas anteriores) de cruce y mutación. Los algoritmos *Simulated Annealing y Genéticos* son algoritmos aleatorios en en sentido de que escogen alternativas a al azar.

Los dos tipos de algoritmos son algoritmos de aproximación: es decir sólo son capaces de encontrar soluciones subóptimas. Es decir soluciones cercanas a la óptima aunque en algunos algoritmos de este tipo es posible demostrar que alcanzan la solución óptima. En muchos casos es posible demostrar que un algoritmo de aproximación encuentra la solución óptima. En otros casos que encuentra una solución cercana al óptimo en un factor ρ. Diremos que es un algoritmo *ρ-aproximado*. Con esto queremos decir exactamente que si tenemos un algoritmo de aproximación de minimización *ρ-aproximado* que encuentra una solución s y s_{op} Es la solución óptima entonces $s \in [s_{op}, \rho s_{op}]$ con $\rho \geq 1$. Si $\rho = 1$ el algoritmo de aproximación es exacto. Para el caso de maximización existe una definición similar con $s \in [\rho s_{op}, s_{op}]$ y $\rho \leq 1$.

1.1 Problemas de optimización

En este capítulo veremos problemas queb pretenden optimizar una función objetivo. Estos problemas pueden escribirse de la forma:

$$\min_{x\in\Omega} f(x)$$

Donde x es un vector de n variables y Ω un dominio dónde las variables deben tomar valores. Los valores de que cumplen $x \in \Omega$ los llamaremos valores válidos. Consideramos que la función objetivo f es una expresión que devuelve valores reales.

Los problemas de maximización pueden transformarse en problemas de minimización de la forma:

$$\max_{x\in\Omega} f(x) = \min_{x\in\Omega} -f(x)$$

En general el dominio Ω se decribirá como un conjunto de restricciones. Algunas de estas son restricciones de desigualdad, de igualdad o de rango. De la forma:

$$g(x) \leq 0$$
$$h(x) = 0$$
$$a \leq x \leq b$$

Dónde a, b son vectores de valores.

Pueden existir otras restricciones que no sean de los tipos anteriores. Pero en un caso general las restricciones pueden ser consideradas como un conjunto de expresiones booleanas construidas con operadores aritméticos o de otros tipos que denominaremos Φ. Visto de esta forma las restricciones pueden ser convertidas a la forma anterior diseñando una función $p_\Phi(x)$ que calculen el número de restricciones incumplidas para cada valor de x. Se trata de añadir al problema la restricción $p_\Phi(x) = 0$ y, por lo tanto, las restricciones se reducen a una de igualdad.

En algunas de las técnicas de optimización que veremos es conveniente reducir el problema de optimización a otro que sólo tenga restricciones de rango. Esto se puede hacer de la forma siguiente. Encontrar una solución al conjunto de restricciones:

$$g_i(x) \leq 0, \quad i = 1, \dots, r$$
$$h_j(x) = 0, \quad j = 1, \dots, s$$
$$a \leq x \leq b$$

Es equivalente al problema de minimización

$$\min_x \sum_{i=1}^{r} (c(g_i(x)))^2 + \sum_{j=1}^{s} (h_j(x))^2$$
$$a \leq x \leq b$$

Dónde $c(z)$ una función d ela forma:

$$c(z) = \begin{cases} 0, & z \leq 0 \\ z, & z > 0 \end{cases}$$

Abordaremos también los llamados problemas multiobjetivo. En estos estos hay que optimizar simultáneamente varios objetivos y por ello el problema es ahora de la forma:

$$\min_{x \in \Omega}(f_1(x), \dots, f_m(x))$$

Este problema puede ser reducido a un problema uniobjetivo de la forma:

$$\min_{x \in \Omega} \sum_{i=1}^{m} \omega_i f_i(x)$$
$$\sum_{i=1}^{m} \omega_i = 1$$

Si uno de los objetivos fuera a maximizar y otro minimizar se trataría de cambiar los signos adecuadamente para que todos los objetivos fueran de minimizar y posteriormente combinarlos. Un caso concreto es cuando queremos incluir en la función objetivo las restricciones del problema. Tenemos las equivalencias:

$$\begin{array}{c} \min\limits_{x} f(x) \\ g_i(x) \leq 0, \;\; i = 1, \dots, r \\ h_j(x) = 0, \;\; j = 1, \dots, s \\ a \leq x \leq b \end{array} \equiv \min_{x} f(x) + R\Big(\sum_{i=1}^{r}(c\big(g_i(x)\big))^2 + \sum_{j=1}^{s}(h_j(x))^2\Big) \quad a \leq x \leq b$$

$$\begin{array}{c} \max\limits_{x} f(x) \\ g_i(x) \leq 0, \;\; i = 1, \dots, r \\ h_j(x) = 0, \;\; j = 1, \dots, s \\ a \leq x \leq b \end{array} \equiv \min_{x} -f(x) + R\Big(\sum_{i=1}^{r}(c\big(g_i(x)\big))^2 + \sum_{j=1}^{s}(h_j(x))^2\Big) \quad a \leq x \leq b$$

Con R un valor suficientemente grande. De forma similar podemos obtener equivalencias que maximicen una función objetivo que incluya la restricciones del problema.

1.2 Codificación de subjuntos de multiconjuntos y sus permutaciones mediante listas

En las técnicas que vemos en este capítulo es necesario codificar soluciones de un problema de una forma que permita, aplicando los operadores adecuados, obtener las posibles soluciones al problema. Vamos a ver aquí la representación de subconjuntos de multiconjuntos y sus posibles permutaciones mediante listas o pares de listas.

Partimos de un multiconjunto (L, M) dónde L es una lista de objetos de tamaño n y M una lista de enteros mayores o iguales a cero, también de tamaño n, indicando $M(i)$ la

multiplicidad del objeto $L(i), 0 \leq i < n$ en el multiconjunto. En el caso de un conjunto $M(i) = 1$. Asociado al problema vamos a definir una lista de índices, que llamaremos lista normal, y representaremos por ln. Para definir esta lista definamos $R(i)$ como la lista formada por $M(i)$ casillas con el valor i entonces ln se define como $ln = R(0) + R(1) + \cdots + R(n-1)$. Las casillas de ln actuarán de índices en la lista L.

Cualquier permutación de un subconjunto de (L, M) puede codificarse mediante un par (b, l) de listas de enteros, cada una de ellas de tamaño $r = |ln\ | = |b| = |l| = \sum_{i=0}^{n-1} m(i)$. La lista b es de ceros y unos. Las casillas de b indicarán si la correspondiente instancia está en al solución. La lista l es una permutación de la lista normal ln que es compartida por todos los subconjuntos de un mismo multiconjunto. Partiendo de la codificación, representada por las listas (b, l), podemos obtener un multiconjunto (o una lista si estamos interesados en el orden) empezando con un multiconjunto vacío (o una lista) y añadiendo una instancia del objeto $L(l(i))$ si $b(i) = 1$. Una permutación de un subconjunto del multiconjunto (L, M) puede codificarse, por lo tanto, por el par de listas (b, l) dónde l es cualquier permutación de la lista ln.

Una permutación de un subconjunto del multiconjunto (L, M) puede codificarse, por lo tanto, por el par de listas (b, l) dónde l es cualquier permutación de la lista ln. Las dos listas (b, l) son del mismo tamaño r y pueden ser reducidas a una lista de tamaño variable pero menor que r que representaremos por d. Esta lista se formará eliminando en l las posiciones tales que la correspondientes en b sean 0. La lista d es otra forma de codificar una permutación de un subconjunto de (L, M). La llamaremos lista decodificada.

Un subconjunto de (L, M), si no estamos interesados en el orden, puede representarse únicamente por la lista b siendo l innecesaria. La información en b puede representarse, también, por una lista m de tamaño n, que cumplen $0 \leq m(i) \leq M(i)$, dónde $m(i)$ representa el número de instancias del objeto $L(i)$ en el subconjunto. A partir de la codificación podemos obtener el multiconjunto asociado iniciando un multiconjunto vacío y para cada i añadiendo $m(i)$ unidades del objeto $L(i)$. A partir de las lista b podemos obtener la m asociada iniciando una lista de tamaño n con todos los valores 0 y para cada i tal que $b(i) = 1$ añadir una unidad a $m(ln\ (i))$.

Un caso particular es cuando queremos obtener permutaciones de un multiconjunto (o conjunto o incluso una lista) fijo. En ese caso la lista b es innecesaria porque siempre estará compuesta de valores 1.

Otro caso particular es cuando queremos obtener subconjuntos de de un multiconjunto pero no estamos interesados en el orden. En este caso la lista l es inecesaria.

Para obtener cambios aleatorios de las listas b, l anteriores necesitamos operadores. Para generar permutaciones de listas de tipo l se usa una lista h, del mismo tamaño, formada por valores reales comprendidos entre cero y uno. Ahora usamos operadores $oh(c, v)$ con $0 \leq c < |l|,\ 0 \leq v \leq 1$ que indican cambiar el valor de la casilla c de h por v. El mecanismo

para generar permutaciones consiste en perturbar h y posteriormente ordenarla de tal forma que cuando cambiamo las casillas i por j en h también las cambiamos en l. Otros operadores de permutación posibles para listas de tipo l son $ol(a,c)$ con $0 \leq a < |l|,\ 0 \leq c < |l|,\ a \neq c$. Aplicar el operador significa permutar los valores de las casillas a, c.

Las listas de tipo m anterior pueden dotarse de un operador del tipo $om(c,v)$, con $0 \leq c < n, 0 \leq v \leq M(c)$, que indica cambiar el contenido de la casilla c por v. Las listas de tipo b son un caso particular de la anteriores y tienen operadores de tipo $ob(c)$, con $0 \leq c < n,$ que indica cambiar el valor de la casilla c.

1.3 Algoritmos de Simulated Annealing

Los *Algoritmos de Simulated Annealing* parten de un problema (como en la estrategia *Voraz*), las soluciones posibles las representamos por un conjunto de estados y para cada uno de ellos un valor calculado por la función objetivo y otro valor que indica si el estado es válido o no. En general asumimos que el conjunto de estados posibles incluye todos los estados que representan soluciones válidas, los estados válidos, y muchos más estados inválidos. Normalmente la función objetivo incluirá términos que penalicen a los estados inválidos como hemos visto arriba.

En este tipo de algoritmos se busca minimizar la función objetivo.

Igual que antes para cada instancia del estado e se definen un conjunto de alternativas A_e. Se escoge de forma aleatoria una de las alternativas de A_e. Se calcula el estado siguiente tras esa alternativa mediante la función $next(e,a)$. Si el estado siguiente se acepta se continúa. Si no se acepta se vuelve al estado anterior.

En este tipo de algoritmos es clave la noción de aceptación del nuevo estado. Sean los estados

$$e' = next(e,a)$$

Sean f, f' los valores de la función objetivo para los estados e, e' y $\Delta = f' - f$ el incremento de la función objetivo. Entonces el nuevo estado se acepta con probabilidad

$$pa(\Delta) = \begin{cases} 1, & \Delta < 0 \\ e^{-\Delta/T}, & \Delta \geq 0 \end{cases}$$

Es decir se acepta con total seguridad si el incremento es negativo (estamos asumiendo problemas de minimización) y con la probabilidad indicada si es positivo. La probabilidad depende de un concepto llamado temperatura. La temperatura, que intenta emular la temperatura de un sistema, toma un valor inicial y posteriormente va disminuyendo hasta acercarse a cero. Una cuestión clave es la estategia de enfriamiento. Es decir el mecanismo disminución de la temperatura. Se han propuesto varias alternativas. Aquí, en un primer momento, escogeremos

$$T = T_0\, \alpha^i, \qquad 0 < \alpha < 1$$

Dónde α es un parámetro, i el número de iteración y T_0 la temperatura inicial. También será necesario establecer n el número de iteraciones y m el número de iteraciones sin cambiar la temperatura.

El algoritmo comienza a dar pasos y en las primeras iteraciones acepta incrementos positivos con probabilidad $e^{-\Delta/T_0}$. Al final acepta incrementos positivos con probabilidad $e^{-\Delta/T_0\alpha^n}$. Para ajustar el algoritmo debemos escoger los parámetros anteriores. Para ello escogemos la probabilidad de aceptación al principio de p_0 y al final p_f. La primera debe ser alta (0.98 por ejemplo) y la segunda baja (0.01 por ejemplo). A partir de lo anterior vemos que debe cumplirse

$$e^{-\Delta/T_0} = p_0, \qquad e^{-\Delta/T_0\alpha^n} = p_f$$

Si conocemos el tamaño típico de Δ y escogemos n (el número de iteraciones con cambio de temperatura en cada una de ellas) podemos despejar T_0, α.

$$T_0 = -\Delta/\ln p_0, \quad \alpha = \sqrt[n]{\frac{-\Delta}{T_0 \ln p_f}}$$

De la relaciones anteriores podemos concluir que T_0 debe ser escogido en función del valor de Δ de tal forma que $\frac{T_0}{\Delta} \cong 100$. El valor de p_0 debe ser cercano a 1 y el de p_f cercano a cero. A partir de las ideas anteriores y las relaciones previas podemos obtener valores para T_0, n, α.

Si $p_0 = 0.99,\ p_f = 0.01, \Delta = 1, n \approx 300$ tenemos $T_0 \approx 100,\ \alpha \approx 0.98$.

O alternativamente

Si $p_0 = 0.99,\ p_f = 0.01, \Delta = 1, n \approx 200$ tenemos $T_0 \approx 100,\ \alpha \approx 0.97$.

Otro parámetro a escoger es m (el número de iteraciones a la misma temperatura). Escogidos esos parámetros el tiempo de ejecución del algoritmo es proporcional al producto $m * n$.

Junto a la anterior expresión para la evolución de la temperatura hay muchas otras posibles. Una de ellas, también bastante común es

$$T = \frac{T_0}{\ln(1 + ai)}$$

Donde i es el número de la iteración, con n iteraciones en total y T_0 la temperatura inicial. Como antes habrá que calcular los valores adecuados de T_0, a, n.

1.4 Algoritmos Genéticos

Los Algoritmos Genéticos se inspiran en la evolución biológica. Estos algoritmos hacen evolucionar una población de individuos sometiéndola a acciones aleatorias semejantes a las que actúan en la evolución biológica (cruces y mutaciones), así como también a una selección de acuerdo con algún criterio, en función del cual se decide cuáles son los individuos más adaptados, que sobreviven, y cuáles los menos aptos, que son descartados. Llamamos generaciones, como en la evolución biológica, a las sucesivas poblaciones que se van obteniendo haciendo evolucionar la primera de ellas.

En este tipo de algoritmos asumimos que se trata de maximizar la función objetivo.

Es esto algoritmos también se estable una condición de parada. El esquema es de la forma:

```
Population evolve(Population initial, StoppingCondition condition) {
	Population current = initial;
	while (!condition.isSatisfied(current)) {
		current = nextGeneration(current);
	}
	return current;
}
```

La población inicial se suele inicializar de forma aleatoria. Cada individuo de la problación (en la estrategia anterior una instancia del estado) tiene una *representación interna* que llamaremos *cromosoma*, codificación o representación del individuo y una *representación externa* (lo que hemos denominado solución en la estrategia anterior) que está más cercana a los individuos tal como se observan el el domino del problema. Para cada problema hay que establecer un mecanismo de decodificación. Es decir una manera de obtener la representación externa de la interna. Cada individuo tiene asociada una media de su fortaleza. Esa medida la denominaremos *fitness*. El objetivo del algortimo genético es encontrar el o los individuos que maximizan su fitnes tras hacer evolucionar la población.

De una forma abstracta un cromosoma podemos verlo como una lista de valores de un tipo dado. Es decir un cromosoma será del tipo *List<T>* con algunas operaciones adicionales. Entre ellas la que calcula la *fitness* del cromosoma y los operadores para hacer la mutación y el cruce. Los elementos de la lista los denominaremos *genes*.

La población es un agregado de individuos que se puede implementar de diferentes formas y que debe tener mecanismos para obtener una generación a partir de otra. Una *población elitista* es aquella que pasa sin cambios un porcentaje de sus mejores individuos a la siguiente generación.

Los mecanismos para obtener la siguiente generación se consiguen aplicando políticas de elitismo, cruce y mutación siguiendo el esquema:

1. Se escogen los mejores individuos de una población sigún la tas de elitismo escogida y se pasan sin copia a a siguiente generación.
2. Se repiten lo siguientes pasos hasta que la nueva generación alcanza el tamaño prefijado.
    a. Siguiendo la *Política de Selección* elegida se escogen dos cromosomas
    b. En un porcentaje establecido por la *Tasa de Cruce* se aplica el *Operador de Cruce* fijado
    c. En un porcentaje establecido por la *Tasa de Mutación* se aplica el *Operador de Mutación* fijado

Una política de selección muy usada es la denomianda *Elección por Torneo*. Consiste en seleccionar sin reemplazamiento un grupo de inviduos al azar de la población y de entre ellos escoger el mejor. El tamaño del grupo se denomina *aridad* del Torneo. Una aridad alta implica que los individuos peores casi nunca son escogidos.

Hay muchas propuestas de operadores de cruce pero los más usuales son:

- *OnePointCrossover:* Se selecciona un punto del cromosoma al azar. La primera parte de cada progenitor se copia en el hijo correspondiente, y la segunda parte se copian en cruz.
- *NPointCrossover:* Se seleccionan N puntos en le cromosoma al azar. El cromosoma queda dividido en N+1 partes. Las primeras partes de cada progenitor se copias en el hijo correspondiente, la segunda parte se copian en cruz, la tercera en el hijo correspondiente, etc.
- *UniformCrossover: Dada una tasa r se escogen r% de genes de un padre y 1-r del otro.* Esto es típicamente un pobre método de cruce, pero la evidencia empírica sugiere que es más exploratorio y resultados en una parte más grande de espacio del problema que se busca.
- *CycleCrossover:* Identificaciclos entre dos cromosomas padres y los copia a los hijos.
- *OrderedCrossover:* Copia un trozos consecutivos de un padre, y completa con los genes restantes del otro padre.

Una documentación de la implementación de estos operadores puede encontarse en apache.

El operador de mutación, con la probabilidad escogida, escoge un gen al azar y lo cambia.

Como hemos dicho arriba los cromosomas podemos representarlos por un tipo que tiene dos propiedades:

- Genes, *List<T>*
- Fitness, Double

Tienen, además, un orden natural definido por fitness. En general *T* podría se cualquier tipo pero aquí sólo vamos a ver con más detalles los casos en que *T* es Double o Integer.

Parámetros de un Algoritmo Genético

Para afinar un algoritmo genético necesitamos, además, dar valores a un conjunto de parámetros de configuración. Algunos de ellos, con la denominación quen usaremos en la implementación son:

- DIMENSION: Dimensión del cromosoma
- POPULATION_SIZE: Tamaño de la población. Usualmente de un valor
- NUM_GENERATIONS: Número de generaciones
- ELITISM_RATE: Tasa de elitismo. El porcentaje especificado de los mejores cromosomas pasa a la siguiente generación sin cambio. Valor usual 0.2
- CROSSOVER_RATE: Tasa de cruce: Indica con qué frecuencia se va a realizar la cruce. Si no hay un cruce, la descendencia es copia exacta de los padres. Si hay un cruce, la descendencia está hecha de partes del cromosoma de los padres. Valores usuales entre 0.8 y 0.95
- MUTATION_RATE: Tasa de mutación. Indica con qué frecuencia serán mutados cada uno de los cromosomas. Si no hay mutación, la descendencia se toma después de cruce sin ningún cambio. La mutación se hace para evitar que se caiga en un máximo local.Valores usales entre 0.5 y 1.
- TOURNAMENT_ARITY: Número de participantes en el torneo para elegir los cromosomas que participarán en el cruce y mutación. Valor usual 2.

Por último es necesario indicar un acondición de parada. Opciones posibles son:

- Tiempo transcurrido. Se acaba cuando pase el timepo indicado
- Numero de generaciones máximo
- Número de soluciones distintas encontradas
- Alguna combiandción de las anteriores

1.5 Programación Lineal Entera

Esta técnica de mucho uso consiste en transformar un problema, normalmente de optimización, en un conjunto de restricciones lineales sobre variables de tipo real más una función objetivo. A este problema transformado los llamaremos *Problema de Programación Lineal*. El problema transformado puede ser resuelto de forma muy eficiente por el conocido algortimo del Simplex.

El problema con esta aproximación consiste en que las variables del problema original pueden tomar valores en dominios discretos como los enteros o valores binarios. El problema transformado no es, por esa razón, exactamente igual al original y por eso se le llama una relajación lineal. En otros casos las restricciones involucradas no son lineales.

El problema relajado linealmente, como hemos comentado, puede resolverse muy eficientemente mediante el Simplex pero con él sólo obtendremos una aproxiamción a la solución del problema.

El problema de programación lineal puede ser completado con otras restricciones como:

- Una variable toma valores enteros en un rango
- Una variable es binaria
- Dado un conjunto de varaibles sólo k de ellas pueden tomar, en la solución, valores distintos de cero.

Un problema de Programación Lineal con restricciones del tipo anterior lo denominaremos *Problema de Programación Lineal Entera*. Para resolver estos problemas se usa una mezcla de relajación lineal más algoritmos de vuelta atrás. Primero se encuentra mediante el algoritmo del Simplex una solución real aproximada y posteriormente mediante vuelta atrás el valor optimo entero o binario y que respete las restricciones adicionales.

Como ejemplo tomemsos el problema de la *Mochila* que hemos resuelto de varias formas. El problema de la *Mochila*, como sabemos, parte de un multiconjunto de objetos $M = (Lo, m)$, dónde Lo es una lista de objetos de tamaño n y m una alista de enteros del mismo tamaño dónde m_i indica el número de repeticiones del objeto en la posición i. A su vez cada objeto ob_i de la lista es de la forma $ob_i = (w_i, v_i)$. Además la mochila tiene una capacidad C. El problema pretende ubicar en la mochila el máximo número de objetos que quepan en la mochila para que el valor de los mismos sea máximo. Si x_i es el número de unidades del objeto i en la mochila el problema puede enunciarse como un problema de Programación Lineal de la forma:

$$\max \sum_{i=0}^{n-1} x_i v_i$$
$$\sum_{i=0}^{n-1} x_i w_i \leq C$$
$$x_i \leq m_i, \quad i \in [0, n-1]$$
$$int\ x_i, \quad i \in [0, n-1]$$

El primer enunciado muestra el objetivo a maximizar. El segundo las restricciones de la capacidad de la mochila. El tercero las debidas al número máximo de unidades disponibles de cada objeto. El último enunciado indica que las variables x_i toman valores en los enteros. Sin este enunciado las varaibles toman valores reales. Tenemos un problema de Programación Lineal Entera o simplemente un Problema de Programación Lineal. El segundo se puede resolver por el algoritmo del Simplex. El primero con una combinación de Simplex y Ramifica y Poda.

Un segundo problema es el problema de la *Asignación*. En este problema tenemos una lsiat de agentes La y una lista de tareas Lt. Ambas del mismo tamaño. El coste de que el agente i

realice la tarea j sea $c(i,j)$. Se pretende asignar a cada agente una tarea y sólo una de tal forma que se ejecuten todas las tareas con el coste mínimo. El problema puede ser escrito como un problema de Programación Lineal Entera de la forma:

Asumimos las variables binarias $x(i,j)$ con valores 1 si el agente i ejecuta la tarea j y cero si no la ejecuta. El problema puede ser plateado de la forma:

$$\min \sum_{i=0,j=0}^{n-1,n-1} x(i,j)\, c(i,j)$$

$$\sum_{i=0}^{n-1} x(i,j) = 1, \quad i \in [0, n-1]$$

$$\sum_{j=0}^{n-1} x(i,j) = 1, \quad j \in [0, n-1]$$

$$bin\, x(i,j), \quad i \in [0, n-1], \; j \in [0, n-1]$$

Ahora las varaibles son binarias, toman valores cero y uno y, de nuevo, tenemos un Problema de Programación Lineal Entera.

2. Detalles de implementación de los Algoritmos de Simulated Annealing

Diseñamos tipos adecaudos para representar los estados de los algoritmos *voraces* y de *simulated annealing*.

```
public interface EstadoSA<E extends EstadoSA<E,S,A>,S,A> {
	E next(A a);
	A getAlternativa();
	S getSolucion();
	boolean condicionDeParada();
	double getObjetivo();
	E copia();
	boolean esSolucion(S s);
}
```

Estos tipos están parametrizados por *S* y *A*. *S* representa el tipo de las soluciones del problema y *A* el tipo de las alternativas posibles. Los métodos tienen el siguiente funcionamiento.

- *E next(A a):* Cambia el estado al siguiente dada una alternativa
- *A getAlternativa():* Escoge una alternativa entre las posibles
- *S getSolucion():* Calcula la solución asociada al estado
- *boolean condicionDeParada():* Implementa la condición de parada.
- *double getObjetivo():* Define el objetivo a minimizar

- *esSolucion(s):* Si el valor s es una solución del problema

Fijados los tipos anteriores podemos implementar el esquema del *Algoritmo de Simulated Annealing.*

```
public class AlgoritmoSA<E extends EstadoSA<E,S,A>,S,A> extends
       AbstractAlgoritmo {

       private ProblemaSA<E,S,A> problema;
       private E estado;
       private E nextEstado;
       public S solucion = null;
       public Set<S> soluciones;

       public E mejorSolucion = null;

       public static Integer numeroMaximoDeIntentos = 10;
       public static Integer numeroDeIteracionesPorIntento = 100;
       public static Integer numeroDeIteracionesALaMismaTemperatura = 10;
       public static Integer numeroDeCambiosAceptados = 0;
       public static Integer numeroDeCambiosNoAceptados = 0;

       public static Integer numeroDeSoluciones = 2;

       public static double temperaturaInicial= 1000;
       public static double alfa = 0.98;

       public AlgoritmoSA(ProblemaSA<E,S,A> p){
              problema = p;
              metricas = Metricas.getMetricas();
              soluciones = Sets.newHashSet();
       }


       private static double temperatura;
       private int numeroDeIteraciones;

       public void ejecuta() {

         mejorSolucion = problema.getEstadoInicial();
         for (Integer n = 0; n < numeroMaximoDeIntentos
                    && soluciones.size() < numeroDeSoluciones; n++) {
              temperatura = temperaturaInicial;
              estado = problema.getEstadoInicial();
              for (numeroDeIteraciones = 0; numeroDeIteraciones <
                    numeroDeIteracionesPorIntento &&
                           !estado.condicionDeParada();
                           numeroDeIteraciones++) {
                    for (int s = 0; s <
                             numeroDeIteracionesALaMismaTemperatura; s++) {
                           A a = estado.getAlternativa();
                           nextEstado = estado.next(a);
                           double incr = nextEstado.getObjetivo()-
                                        estado.getObjetivo();
                           if (aceptaCambio(incr)) {
                                  estado = nextEstado.copia();
```

```
                                actualizaMejorValor();
                        }
                        nexTemperatura();
                }
                solucion = estado.getSolucion();
                if (solucion != null)
                        soluciones.add(solucion);
        }
    }

    private void nexTemperatura() {
        temperatura = alfa*temperatura;
    }

    private boolean aceptaCambio(double incr) {
        return Math2.aceptaBoltzmann(incr, temperatura);
    }

    private void actualizaMejorValor() {
        if (estado.getObjetivo() < mejorSolucion.getObjetivo()) {
            mejorSolucion = estado.copia();
        }
    }
}
```

Una versión actualizada puede encontrarse en el API.

Como podemos ver el algortimo implemena dos estrategias: *Voraz* y *Simulated Annealing*. En el caso de la estrategia *Voraz* el algoritmo escoge una alternativa y pasa al estado siguiente. Repite estos pasos hasta que encuentre le criterio de parada o cambie de estrategia.

En el caso de *simulated annealing* escoge una alternativa y decide si aceptar el cambio o no. Lo acepta o no según la estrategia explicada anteriormente. Para implementar la aceptación o el rechazo del cambio necesitamos hacer una copia del estado para posteriormente confirmar el cambio o restaurar el estado anterior.

La aceptación o no depende de la temperatura que va disminuyendo progresivamente. Hay diferentes posibilidades de ir enfriando la temperatura. Algunas de ellas se han inclouido en el código. Para cada temperatura se hacen un número de iteraciones a temperatura constante y se escoge le mejor valor obtenido para pasarlo a iteración con el siguiente valor de la temperatura.

3. Algoritmos Genéticos detalles de implementación

En el API incluimos una implementación de los *Algoritmos Genéticos*. La implementación es una adaptación del software que se ofrece en Apache.

Aquí solo incluimos algunos detalles para dar una idea de la implementación.

Es esque del Algoritmo Genético es de la forma:

```
public Population evolve(Population initial, StoppingCondition condition) {
	Population current = initial;
	while (!condition.isSatisfied(current)) {
		current = nextGeneration(current);
	}
	return current;
}
```

Los tipos siguientes son ofrecidos en Apache. CrossoverPolicy:

- *CrossoverPolicy:* Operador de cruce
- *MutationPolicy:* Operador de mutación
- *SelectionPolicy:* Operador de selección
- *StoppingCondition:* Condición de parada
- *Population:* Agregado de Chromosome.
- *Chromosome: Un cromosoma*.

Cualquier problema que quiera ser resuelto mediante Algoritmos Genéticos debe implementar el tipo siguiente donde:

- S es el tipo de la solución
- C el tipo de los cromosomas

```
public interface ProblemaAG<S, C extends Chromosome> {

	C getInitialChromosome();
	S getSolucion(C chromosome);
}
```

Para concretar un problema debemos instanciar el tipo S con el tipo de la solución, y el tipo C con el tipo del cromosoma que a su vez debe heredar de una de las clases *BinaryChromosome, RandomKey<T> o MixChromosome.*

Con esos tipos el esquema de ejecución de un Algoritmo Genético (método ejecuta) puede implementarse en la clase AlgoritmoAG.

El código completo puede encontrarse en AlgoritmoAG. Algunos detalles son:

```
CrossoverPolicy crossOverPolicy;
MutationPolicy mutationPolicy;
SelectionPolicy selectionPolicy;

StoppingCondition stopCond;
```

```
Population initialPopulation;
Population finalPopulation;
Chromosome bestFinal;

ElitisticListPopulation randomPopulation() {
	List<Chromosome> popList = new LinkedList<>();

	for (int i = 0; i < POPULATION_SIZE; i++) {
		Chromosome randChrom = getCromosome();
		popList.add(randChrom);
	}
	return new ElitisticListPopulation(popList,popList.size(),ELITISM_RATE);
}

Chromosome getCromosome() {…};

void ejecuta() {
	GeneticAlgorithm ga = new GeneticAlgorithm(
		crossOverPolicy,
		CROSSOVER_RATE,
		mutationPolicy,
		MUTATION_RATE,
		selectionPolicy);
	finalPopulation = ga.evolve(initialPopulation, stopCond);
	bestFinal = finalPopulation.getFittestChromosome();
}
```

3.1 Codificación mediante cromosomas de subconjuntos de multiconjuntos y sus permutaciones

Como hemos dicho cada cromosoma tiene asociada una lista de genes de tamaño r, la dimensión del cromosoma, que contiene los genes. Un cromosoma es un tipo de la forma:

```
public interface Cromosoma<T> {
	List<T> decode();
	double fitness();
}
```

La información de un problema cuya solución es un subconjunto de un multiconjunto o una permutación del mismo se puede recoger en el tipo:

```
public interface ProblemaAGBag<S> {
	Integer getNumeroDeObjetos();
	Integer getMultiplicidadMaxima(int index
	Double fitnessFunction(List<Integer> ls);
	S getSolucion(Cromosoma<Integer> chromosome);
}
```

Dada la información proporcionada en en tipo *ProblemaAGBag* es posible construir una secuencia normal, tal como explicado en el apartado 1.2. Dada una secuencia normal, ln, podemos codificar un subconjunto de un multiconjunto o una permutación del mismo mediante las listas (b, l, d) tal como vimos. Asumimos que el tamaño de la secuencia normal es r y el número de objetos n.

Para concretar esas ideas implementamos cuatro cromosomas:

- *BagBinaryChromosome:* Su lista decodificada está formada por una lista de tamaño menor o igual a r cuyos valores son índices en el rango $[0, n-1]$, y cada índice i se puede repetir un máximo número de veces dado por la multiplicidad máxima del objeto definida en el problema. La implementación usa un cromosoma binario del tamaño de la secuencia normal. Es un cromosoma adecuado para codificar problemas de subconjuntos de multiconjuntos. Es una implementación del par de listas (ln, b) vistas arriba.
- *BagRandomKeyChromosome:* Su lista decodificada está formada por una lista de tamaño r, cuyos valores son índices en el rango $[0, n-1]$, y cada índice i se repite un número de veces dado por la multiplicidad máxima del objeto definida en el problema. La implementación usa un cromosoma de clave aleatoria de tamaño r. Es un cromosoma adecuado para codificar problemas de permutaciones. Es una implementación del par de listas (ln, l) vistas arriba.
- *BagMultiChromosome: Su* lista decodificada está formada por una lista de tamaño n cuyos elementos para cada i son valores en en rango *[0,m(i)],* siendo *m(i)* la multiplicidad máxima para i. La implementación usa un cromosoma binario del tamaño n*nbits. Siendo nbits el número de bits usados para representar cada uno de los enteros. Es un cromosoma adecuado para codificar problemas de subconjuntos de multiconjuntos. Es una implementación de la lista m.
- *BagMixChromosome:* La lista decodificada está formada por una lista de tamaño menor o igual que r, cuyos valores son índices en el rango [0,n-1], y cada índice i se puede repetir un máximo número de veces dado por la multiplicidad máxima del objeto i definida en el problema. La implementación usa un cromosoma binario y otro de tipo clave aleatoria. Ambos de tamaño r. Es un cromosoma adecuado para codificar problemas de permutaciones de subconjuntos de multiconjuntos. Es una implementación de la combinación de listas (ln, b, l).

En la clase Algoritmos se incluyen métodos de factoría para crear algoritmos que usen los cromosomas anteriores:

```
static AlgoritmoAGBinary createAGBinary(ProblemaAGBag<?> p) {...}
static AlgoritmoAGMulti createAGMulti(ProblemaAGBag<?> p) {... }
static <S> AlgoritmoAGRandomKey createAGRandomKey(ProblemaAGBag<?> p){... }
static AlgoritmoAGMix createAGMix(ProblemaAGBag<?> p) {...}
```

3.2 Codificación de números reales mediante cromosomas binarios

- Los cromosomas binarios pueden usarse para reprentar valores de muchos tipos. Por ejemplo para representar los número reales en el intervalo $[x, y]$ con un precisión dada.

$$\Gamma(e) = x + \frac{y - x}{2^n - 1} e$$

Dónde e el entero cuya representación en binario es la información el el cromosoma y n la dimensión de esa lista. Como se puede ver sólo se obtienen 2^n número reales.
Los cromosomas binarios también pueden usarse para respresentar valores en el hipercubo de R^k definido por las cotas $[x_i, y_i]$, $i \in [0, k)$. Usaremos listas binarias de dimensión nk y la docodificación será de la forma

$$\Gamma(e) = \Gamma(e_0, \dots, e_{k-1}) = [x_i + \frac{y_i - x_i}{2^n - 1} e_i]$$

Donde e_i es el entero que viene representado en binario por la sublista $ls(ni, ni + n)$ de la lista binaria ls asociada al cromosoma.

El cromosoma anterior está implementado en:

- *RealBinaryChromosome*

La información para usar este cromosoma se recoge en el tipo:

```
public interface ProblemaAGReal<S> {
	Integer getNumeroDeVariables();
	List<Par<Double,Double>> getLimites();
	Double fitnessFunction(List<Double> ls);
	S getSolucion(Cromosoma<Double> chromosome);
}
```

Y en la clase Algoritmos se incluye una factoría para usar estos cromosmas.

```
public static AlgoritmoAGReal createAGReal(ProblemaAGReal<?> p) {...}
```

Junto a las anteriores formas de codificación hay muchas otras que pueden verse en la literatura.

3.3 Ejemplos

Ejemplos de clase de Test para el Problema de la Mochila es:

```
public static void main(String[] args){
		AlgoritmoAG.NUM_GENERATIONS = 10000;
		AlgoritmoAG.ELITISM_RATE  = 0.30;
		AlgoritmoAG.CROSSOVER_RATE = 0.8;
		AlgoritmoAG.MUTATION_RATE = 0.7;
		AlgoritmoAG.POPULATION_SIZE = 100;
		AlgoritmoAG.SOLUTIONS_NUMBER = 1;
		AlgoritmoAG.FITNESS = 623;
		AlgoritmoAG.stoppingConditionType =
			AlgoritmoAG.StoppingConditionType.SolutionsNumber;

		AlgoritmoAG.crossoverType = AlgoritmoAG.CrossoverType.OnePoint;

		ProblemaMochila.capacidadInicial = 78;
		ProblemaAGBag<SolucionMochila> p =
			new ProblemaMochilaAG("objetosmochila.txt");
		AlgoritmoAGBinary<SolucionMochila> ap =
			Algoritmos.createAGBinary(p);
		ap.ejecuta();


		System.out.println("================================");
		System.out.println(
			p.getSolucion((BagChromosome) ap.getBestFinal()));
		System.out.println("================================");
		System.out.println(AlgoritmoAG.BestChromosomes.size());

}
```

El código completo para éste ejemplo puede verse en el paquete Mochila.

Y para el Problema de las Reinas:

```
public static void main(String[] args){
		AlgoritmoAG.NUM_GENERATIONS = 150000;
		AlgoritmoAG.ELITISM_RATE  = 0.20;
		AlgoritmoAG.CROSSOVER_RATE = 0.8;
		AlgoritmoAG.MUTATION_RATE = 0.8;
		AlgoritmoAG.POPULATION_SIZE = 400;
		AlgoritmoAG.SOLUTIONS_NUMBER = 1;
		AlgoritmoAG.FITNESS = 0.;
		AlgoritmoAG.stoppingConditionType =
			AlgoritmoAG.StoppingConditionType.SolutionsNumber;
		AlgoritmoAG.crossoverType = AlgoritmoAG.CrossoverType.OnePoint;
		ProblemaReinasAG.numeroDeReinas = 20;
		ProblemaAGBag<List<Reina>> p = ProblemaReinasAG.create();
		AlgoritmoAGRandomKey<List<Reina>> ap =
			Algoritmos.createAGRandomKey(p);
		ap.ejecuta();
		System.out.println("================================");

		System.out.println("================================");
```

```
            for (Chromosome c: AlgoritmoAG.BestChromosomes) {
                  System.out.println(p.getSolucion((BagChromosome) c));
            }
            System.out.println("================================");
            System.out.println(AlgoritmoAG.BestChromosomes.size());
}
```

El código completo para este ejemplo puede verse en el paquete Reinas.

Y para el problema de los anuncios visto anteriormente.

```
public static void main(String[] args){

            AlgoritmoAG.NUM_GENERATIONS = 400;
            AlgoritmoAG.ELITISM_RATE  = 0.30;
            AlgoritmoAG.CROSSOVER_RATE = 0.8;
            AlgoritmoAG.MUTATION_RATE = 0.7;
            AlgoritmoAG.POPULATION_SIZE = 100;
            AlgoritmoAG.SOLUTIONS_NUMBER = 1;
            AlgoritmoAG.FITNESS = 623;
            AlgoritmoAG.stoppingConditionType =
                  AlgoritmoAG.StoppingConditionType.GenerationCount;

            ProblemaAnuncios.tiempoTotal = 30;

            ProblemaAGBag<ListaDeAnunciosAEmitir> p =
                  new ProblemaAnunciosAG("anuncios.txt");
            AlgoritmoAGMix<ListaDeAnunciosAEmitir> ap =
                  Algoritmos.createAGMix(p);
            ap.ejecuta();


            System.out.println("================================");
            System.out.println(
                  p.getSolucion((BagChromosome) ap.getBestFinal()));
            System.out.println("================================");
      }

}
```

El código completo para este ejemplo puede verse en el paquete Anuncios.

Para resolver problemas medaite algoritmos genéticos debemos implementar los interfaes siguientes dependiendo del tipo de problema:

```
public interface ProblemaAG {

      int getDimension();
}
public interface ProblemaAGBag<S> extends ProblemaAG {

      List<Integer> getNormalSequence();
```

```
        Double fitnessFunction(List<Integer> ls);
        S getSolucion(BagChromosome chromosome);
}
public interface ProblemaAGReal<S> extends ProblemaAG {

        Integer getNumeroDeVariables();
        Integer getItemsPorVariable();
        List<Par<Double,Double>> getLimites();
        Double fitnessFunction(List<Double> ls);
        S getSolucion(RealBinaryChromosome chromosome);

}
```

Los métodos tienen el siguiente significado

- *S*: Tipo de la solución
- *int getDimension():* La dimensión del cromosoma
- *List<Integer> getNormalSequence():* La secuencia normal asociada al problema
- *Double fitnessFunction(List<Integer> ls):* La function de fitness asociada a la lista de indices.
- *S getSolucion(BagChromosome chromosome):* La solución asociada al cromosoma
- *Integer getNumeroDeVariables():* Número de variables
- *Integer getItemsPorVariable():* Número de bits para representar cada variable real
- *List<Par<Double,Double>> getLimites():* Limites por cada variable que definen el dominio del problema

4. Programación Lineal Entera. Detalles de Implementación

Para modelar los problemas de Programación Lineal Entera o no usaremos el tipo IProblemaPL y la clase ProblemaPL que lo implementa.

```
public interface IProblemaPL {

        public TipoDeOptimizacion getTipo();
        public LinearObjectiveFunction getObjectiveFunction();
        public void setObjectiveFunction(LinearObjectiveFunction
                        objectiveFunction);
        public void setObjectiveFunction(double[] coefficients, double
                        constantTerm);
        public Collection<LinearConstraint> getConstraints();
        public void addConstraint(LinearConstraint lc);
        public void addConstraint(double[] coefficients, Relationship
                relationship, double value);
        public void addSosConstraint(List<Integer> ls, Integer nv);
        public Integer getGetNumOfVariables();
        public void setTipoDeVariable(int e, TipoDeVariable tipo);
        public void setTipoDeTodasLasVariables(TipoDeVariable tipo);
        public void setVariableLibre(int e);
        public void setVariableSemicontinua(int e);
        public List<Integer> getVariablesEnteras();
```

```
	public List<Integer> getVariablesBinarias();
	public void setNombre(Integer e, String s);
	public String getNombre(Integer e);
	public List<Integer> getVariablesLibres();
	public List<Integer> getVariablesSemicontinuas();
	public String toStringConstraints();
	public void toStringConstraints(String fichero);
}
```

La descripción más detallada de los métodos puede encontrarse en clase ProblemaPL.

Una vez construido el problema mediante le tipo anterior disponemos de dos algoritmos para buscar las soluciones:

- AlgoritmoPL. Que implementa el método del Simplex reutilizando las librerías de Apache.
- AlgoritmoPLI. Que implementa un algoritmo de Programación Lineal Entera reutilizando las librerías de LpSolve.
- Algoritmos. La factoría anterior dispone de los métodos siguientes:

```
	public static AlgoritmoPL createPL(ProblemaPL p) {
		return AlgoritmoPL.create(p);
	}
	public static AlgoritmoPLI createPLI(String fichero) {
		return AlgoritmoPLI.create(fichero);
	}
```

Con el primero a partir de un objeto del tipo *ProblemaPL* se instancia un algoritmo del Simplex para resolverlo. Se ignoran todas las restricciones que no sean específicas de Progrmación Lineal. Por ejemplo la declaración de variables enteras o binarias.

El segundo método toma un fichero de entrada escrito en el formato adecuado para ser procesado por LpSolve e instancia un algoritmo para resolverlo. El formato de este tipo de fichero puede verse en formato LpSolve. Pero también puede genrarse mediante el método:

```
public void toStringConstraints(String fichero);
```

Del tipo *ProblemaPL* visto antes.

En el código siguiente puede verse un aplicación para el caso de la Mochilla. Este código puede encontrarse en MochilaPL.

```
public class MochilaPL {

	public static SolucionMochila getSolucion(double[] d){
		SolucionMochila s = SolucionMochila.create();
		for (int i = 0; i <
			ProblemaMochila.getObjetosDisponibles().size(); i++) {
			s = s.add(ProblemaMochila.getObjeto(i), (int)
```

```
                        Math.round(d[i]));
            }
            return s;
      }

      public static ProblemaPL getProblemaPL() {
            ProblemaMochila.leeObjetosDisponibles("objetosMochila.txt");
            ProblemaMochila.capacidadInicial = 78;
            int num = ProblemaMochila.getObjetosDisponibles().size();
            ProblemaPL p = ProblemaPL.create(num,
                  ProblemaPL.TipoDeOptimizacion.Max);
            double [] d = Arrays2.getArrayDouble(num, 1.);
            for (int i = 0; i < num ; i++) {
                  d[i] = ProblemaMochila.getValorObjeto(i);
            }
            p.setObjectiveFunction(d, 0.);
            d = Arrays2.getArrayDouble(num, 0.);
            for (int i = 0; i < num ; i++) {
                  d[i] = ProblemaMochila.getPesoObjeto(i);
            }
            p.addConstraint(d, Relationship.LEQ,
                  ProblemaMochila.capacidadInicial);
            for (int i = 0; i <
                    ProblemaMochila.getObjetosDisponibles().size(); i++) {
                  d = Arrays2.getArrayDouble(num, 0.);
                  d[i] = 1.;
                  p.addConstraint(d, Relationship.LEQ,
                    ProblemaMochila.getNumMaxDeUnidades(i));
            }
            return p;
      }


      public static void main(String[] args) {
            ProblemaPL p = getProblemaPL();
            int num = ProblemaMochila.getObjetosDisponibles().size();
            AlgoritmoPL a = Algoritmos.createPL(p);
            a.ejecuta();
            System.out.println(p.toStringConstraints());
            for (int i = 0; i < num; i++) {
                  System.out.println("Num Unidades de "+
                  ProblemaMochila.getObjeto(i).getCodigo()+ " =
                  "+a.getSolutionPoint()[i]+" Num Max ="
                  +ProblemaMochila.getNumMaxDeUnidades(i));
            }
            System.out.println("________");
            System.out.println(a.getSolutionValue());
            SolucionMochila s = MochilaPL.getSolucion(a.getSolutionPoint());
            System.out.println("________");
            System.out.println(s);
      }

}
```

5. Comparación entre las técnicas: un ejemplo

Veamos un par de ejemplos para concretar las ideas. En primer lugar veremos el problema de los anuncios un problema de otpimización. En segundo lugar el problema de las n-reinas. Un problema de búsqueda de una o varias soluciones que reducimos a un problema de optimización.

Estos problemas pueden resolverse, junto con las técnicas que estamos viendo en este capítulo, con las que hemos visto en capítulos precedentes.

5.1 Problema de Anuncios de Televisión

Enunciado del problema:

Un canal de televisión quiere obtener el máximo rendimiento (en euros) de la secuencia de anuncios que aparecerá en la cadena después de las campanadas de fin de año. Dicho canal ha calculado que la primera secuencia de anuncios del año durará *T* segundos como máximo, antes que empiece la presentación del programa de fin de año. Para cubrir dicha secuencia de anuncios el canal ha recibido *N* ofertas de anuncios existiendo restricciones para la emisión entre unos anuncios y otros. Por ejemplo:

Cliente:	1	2	3	4	5	6	7
Tiempo Anuncio (segundos):	5	25	25	5	15	10	7
Oferta (miles de euros):	4	10	20	3	15	8	10

Debido a que existen empresas que compiten con productos similares, se han establecido las siguientes restricciones en las cláusulas de los contratos:

- Si aparece el Anuncio 1 no puede aparecer el Anuncio 4 en la secuencia y viceversa
- Si aparece el Anuncio 3 no puede aparecer con el Anuncio 6 en la secuencia y viceversa

El precio final del anuncio (*pfa*) dependerá de la posición que éste ocupe dentro de la secuencia, su cálculo se realizará utilizando la siguiente fórmula:

$$pfa(oferta, pos) = oferta * \frac{1000}{pos} + 50000\ (euros)$$

Dónde *pos* es el segundo en que comienza el anuncio asumiendo que los segundos se numeran desde 1.

Se pide diseñar un algoritmo *Voraz*, otro de *Ramifica y Poda* y un tercero que use *Simulated Annealing* para resolver le problema.

Veamoos en prime lugar una estrategia *Voraz*.

Solución:

Cada anuncio podemos modelarlo como:

- *Duración: Double*, Consultable
- *Cliente: Integer*, Consultable
- *Oferta: Double*, Consultable (precio ofrecido por el cliente)
- *PrecioFinal(Oferta,Posicion), Double*, Consultable, Se calcula por la fórmula $PrecioFinal = Oferta\frac{1000}{Posicion} + 50000$
- *PrecioUnitario, Double,* Consultable, Se calcula por Oferta/Duración

Propiedades del Problema

Compartidas

- *LA, AnunciosDisponibles: List<Anuncio>.* Anuncios disponibles
- *T, TiempoTotal: Double*
- *R, Restricciones, Map<Integer,Set<Integer>>,* restricciones para un anuncio dado.

Propiedades básicas del estado

- *AE, AnunciosAEmitir: List<Anuncio>. Anuncios ya decididos a emitir*

Propiedades Derivadas del Estado

- *TC, TiempoConsumido: int.* Suma de la duración de los tiempos de los anuncios a emitir.
- *TR, TiempoRestante: Double.* Diferencia entre el tiempo total y el consumido.
- *APE, AnunciosParaEscoger: List<Anuncio>.* Lista de anuncios restantes que no tienen incompatibilidades con los ya decididos y cuya duración es menor que el tiempo restante. Ordenada de mayor a menor según su precio unitario.
- *V, Valor: Double.* Valor de los anuncios ya decididos.

Conjunto de Alternativas: Solo consideramos la posibilidad de añadir al final de los anuncios a emitir el primero de los anuncios a escoger.

Dadas las propiedades básicas es posible calcular las propiedades derivadas. El estado queda concretado cuando fijamos las propiedades básicas.

Problema anuncios de televisión	
Técnica: Voraz	
Propiedades Compartidas	*LA: List<Anuncio>, ordenada por Precio Unitario.* *T: Double, Tiempo total* *R: Map<Integer,Set<Integer>>, restricciones entre anuncios*
Propiedades básicas del Estado	*AE: List<Integer>, posiciones en LA de los anuncios a emitir*
Propiedades derivadas del Estado	*TC: Double, tiempo consumido*

	TR: Double, tiempo restante *APE: List<E>, anuncios para escoger* *V: Double, valor*
Solución: SolucionAnuncios	
Alternativas: Solo hay una alternativa consistente en escoger el primero de APE si es no vacío	
Elección: Escoge la alternativa dada.	
Estado inicial: (Ø)	
Estado final: AE=Ø	
next(a): *(AE+a)*	

El algoritmo anterior es un algoritmo voraz. Por lo tanto, en un caso general, no nos asegura que la solución sea óptima. Para conseguir la solución óptima debemos usar un algoritmo de *Backtraking* o uno de *Ramifica y Poda* si tenemos disponible una cota superior para el valor obtenido mediante una solución. Vamos a escoger esta segunda aproximación asumiendo que podemos disponer de una función de cota.

Una posible función de cota vendría dada por el valor obtenido al emitir todos los anuncios en APE por orden hasta que se consuma el tiempo y permitiendo que el último que se emitiera se pudiera emitir sólo en parte.

Ahora las alternativas son todos los anuncios en *APE.*

Problema anuncios de televisión	
Técnica: Ramifica y Poda	
Propiedades Compartidas	*LA: List<Anuncio>, ordenada por Precio Unitario.* *T: Double, Tiempo total* *R: Map<Integer,Set<Integer>>, restricciones entre anuncios* *VM, Double, Valor Máximo*
Propiedades básicas del Estado	*AE: List<Integer>, posiciones en LA de los anuncios a emitir*
Propiedades básicas del Estado	*TC: Double, tiempo consumido*

	TR: Double, tiempo restante *APE: List<E>, anuncios para escoger* *V: Double, valor*
Solución: SolucionAnuncios	
Alternativas: $A = \{a: APE\}$	
Estado inicial: ($\emptyset$)	
Estado final: $APE = \emptyset$	
Add(a): (AE)->(AE+a)	
Remove(a): (AE)->(AE-a)	

Por ultimo podemos usar la técnica de *Simulated Annealing*. Para usar esta técnica la primera decisión a tomar es la forma de codificar el estado y a partir de ahí las alternativas posibles. Las alternativas más su correspondiente cambio de estado son similares a los operadores de mutación de los algoritmos genéticos. Tenemos dos enfoques generales:

- Codificar los estados del tal manera que sólo representen estados válidos, estados que cumplan las restricciones, y a partir de ahí operadores de mutación (alternativas más cambios de estado) que poduzcan estados válidos. Este enfoque suele requerir operadores específicos para cada problema y una comprobación de que el estado que representa el valor óptimo es alcanzable dese el estado inicial a través de los operadores de mutación propuestos. En este caso los operadores de mutación podrían ser tres: añadir, eliminar e intercambiar. Estos operadores los podemos representar de la forma:$(a, p1, p2)$, (r, p) y $(i, p1, p2)$. El primer tipo representa el añadir en la posición $p1$ de *AE* el anuncio que se encuentre en la posición $p2$ de *APE*, la segunda consiste en eliminar el anuncio en la posición p de *AE* y la tercera en intercambiar los anuncios en als posiciones $p1, p2$. Como podemos comprobar estas operaciones no estados por lo que hay que filtrar las operaciones disponible en cada estado. Este enfoque, aunque posible, no es recomendable.
- Un segundo enfoque más recomendable es usar una codificación y unos operadores de mutación similares a los correspondientes en los algoritmos genéticos. Para modelar el problema mediante Algoritmos Genéticos usamos el cromosoma mixto por lo tanto el estado estaría dotado de dos listas de tamaño n, el número de anuncios. La primera lista de ceros y unos. La segunda formada por una permutación de los valores del conjunto $[0, n-1]$. Los operadores de mutación para ambas listas los hemos visto más arriba. Las alternativas más los correspondientes cambios de estado están formadas por un operador de mutación para una de las listas anteriores. Como podemos comporbar ahora representamos individuos válidos, que cumplen las

restricciones, e individuos no válidos. Por lo tanto tenemos que introducir las restrcciones en al función objetivo tal como hemos esxplciado arriba.

La ficha del problema siguiendo la primera opción es:

Problema anuncios de televisión	
Técnica: Simulated Annealing	
Propiedades Compartidas	*LA: List<Anuncio>, ordenada por Precio Unitario.* *T: Double, Tiempo total* *R: Map<Integer,Set<Integer>>, restricciones entre anuncios*
Propiedades básicas del Estado	*AE: List<Integer>, posiciones en LA de los anuncios a emitir*
Propiedades básicas del Estado	*TC: Double, tiempo consumido* *TR: Double, tiempo restante* *APE: List<E>, anuncios para escoger* *V: Double, valor*
Solución: SolucionAnuncios	
Objetivo: V	
Alternativas: $A1 = \{(a, p1, p2): p1 \in [0, \lvert AE \rvert],\ p2 \in [0, \lvert APE \rvert)\}$ $A2 = \{(r, p): p \in [0, \lvert AE \rvert)\}$ $A3 = \{(i, p1, p2): p1, p2 \in [0, \lvert AE \rvert)\}$ $A = A1 \cup A2 \cup A3$	
Estado inicial: ($\emptyset$*)*	
next($(a, p1, p2)$*): (AE)->Se añade el anuncio en la posición p2 de APE en la posición p1* *next(*(r, p)*): (AE)->Se elimina el anuncio en posición p* *next(*$(i, p1, p2)$*): (AE)->Se intercambian los anuncios en las posiciones p1, p2*	

La versión mediante algoritmos genéticos puede verse en el paquete Anuncios mediante Genéticos.

La versión mediante Vuelta Atrás puede verse en el paquete Anuncios mediante Vuelta Atrás.

La estrategia Voraz está conmsiderada en el diseño de la versión de Vuelta Atrás con filtro considerada.

La versión mediante Simulated Annealing, siguiendo el primer enfoque, puede verse en el paquete Anuncios mediante Simulated Annealing.

5.2 Problema de las n-Reinas

El problema de las Reinas ya lo hemos modelado en temas anteriores. Como vimos allí el problema se enuncia así:

Colocar Nr reinas en un tablero de ajedrez $Nr \times Nr$ de manera tal que ninguna de ellas amenace a ninguna de las demás. Una reina amenaza a los cuadrados de la misma fila, de la misma columna y de las mismas diagonales. Las filas y columnas toman valores en $(0, Nr-1)$.

Ahora las propiedades del estado son:

Nr: Número de reinas

Fo, lista de enteros, Filas ocupadas por la reinas

Dp, Multiconjunto de enteros, Diagonales principales ocupadas

Ds, Multiconjunto de enteros, Diagonales secundarias ocupadas.

Problema de las Reinas	
Técnica: Simulated Annealing	
Propiedades Compartidas	*Nr, entero*
Propiedades básicas del estado	*Fo, lista de enteros*
Propiedades derivadas del estado	*Dp, Conjunto de enteros* *Ds, Conjunto de enteros*
Solución: List<Reina>	
*Objetivo: 2*Nr-|Dp|-|DS|*	

Inicial: Fo = [0,1,2,…,Nr-1]
Alternativas: $A = \{(p1, p2):\ p2 > p2,\ \ p1 \in [0, \lvert Fo \rvert),\ \ p2\ \in [0, \lvert Fo \rvert)\}$
$next\big((p1, p2)\big) =$ Se intercambian los valores en als posiciones dadas

Dónde por *|Dp|, |Ds|* hemos representado el número de objetos distintos en cada uno d elos multiconjuntos.

Problema de las Reinas	
Técnica: Algoritmos Genéticos. RandomKey	
Propiedades Compartidas	*Nr, entero*
Propiedades básicas del individuo	*Fo, lista de enteros*
Propiedades derivadas del individuo	*Dp, Conjunto de enteros* *Ds, Conjunto de enteros*
Solución: List<Reina>	
*Objetivo: 2*Nr-|Dp|-|DS|*	
Inicial: Fo = [0,1,2,…,Nr-1]	
Codificación del Cromosoma: Double	
Decodificación: Según la técnica de RandomKey	