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 en primer lugar la *Programación Lineal Entera* que es una técnica para encontrar soluciones exactas a algunos problemas de optimización o aproximadas relajando algunas de las restricciones impuestas.

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 $\rho \leq 1$.

2. Programación Lineal Entera

Esta técnica consiste en modelar un problema, normalmente de optimización, en un conjunto de restricciones lineales sobre variables de tipo real más una función objetivo a maximizar o minimizar. A este problema transformado los llamaremos *Problema de Programación Lineal*. El problema modelado así puede ser resuelto de forma muy eficiente por el conocido *Algoritmo del Simplex* que puede resolver el problema en tiempo polinomial en la gran mayoría de los casos.

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

- Algunas variables toman valores enteros
- Algunas variables toman valores binarios
- Dado un conjunto de variables 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*. Esta metodología tiene un mayor poder expresivo que la Programación Lineal pero su complejidad computacional es más alta que polinomial. Si eliminamos las restricciones de tipo entero y binario obtenemos un Problema de Progrmación Lineal eficiente de resolver pero que no es exactamente igual al original. Al problema

resultante se le llama relajación lineal del primero y su solución es una aproximación del mismo.

Para resolver estos problemas se usa una mezcla de relajación lineal más algoritmos de vuelta atrás. El problema relajado linealmente, como hemos comentado, puede resolverse muy eficientemente en tiempo polinomial mediante el algoritmo del Simplex pero con él sólo obtendremos una aproximación a la solución del problema de Programación Lineal Entera. Para resolver este primero se encuentra, mediante el algoritmo del Simplex, la solución de la relajación lineal que es aproximada y posteriormente, mediante vuelta atrás, el valor optimo entero o binario y que respete las restricciones adicionales. Los algoritmos que resuelven los problemas de Programación Lineal Entera tienen comlejidades superiores a las polinomiales.

Aquí, por el momento estamos interesados en los problemas de modelado mediante Progrmación Lineal Entera asumiendo que disponemos de los algoritmos necesarios para encontrar la solución. Para concretar un Problema de *Programación Lineal Entera* se compone de:

- Un conjunto de variables reales
- Un conjunto de varibles pueden tomar valores enteros
- Un conjunto de variables pueden tomar valores binarios
- Una función objetivo compuesto como una combinación lineal de las varaibles
- Un conjunto de restricciones (mayor, menor, igual, mayor o igual, menor o igual) entre combinaciones lineales de variables

Adicionalmente se pueden añadir otro tipo de variables libres, semicontinuas y restricciones SOS que veremos más adelante. Veamos ejemplos y detalles de este tipo de modelos.

2.1 Un catálogo de problemas

Veamos aquí un catálogo de problemas que pueden ser resueltos mediante modelos de Programación Lineal Entera o Mixta.

Problemas de redes de flujo

De forma compacta los problemas de redes de flujo se pueden modelar, como hemos visto en capítulos anteriores, como grafos dirigidos con un conjunto de vértices fuente (F), un conjunto de vértices sumidero (S), y un conjunto de restricciones sobre los flujos producidos, los consumidos, lso flujos circulantes por los vértices y por las aristas. Asumamos que V es el conjunto de vértices, F el conjunto de vértices fuente, S el conjunto de vértices sumidero, $V - F \cup S$ el conjunto de vértices intermedios, E el conjunto de aristas, w_j el costo unitario asociado al paso del flujo por la $j \in E$, el costo unitario asociado a la producción o consumo o paso de una unidad por el vértices $i \in V$ lo representaremos por w_i, a las cotas superiores e inferiores en vértices y aristas los representamos respectivamente por $b_i^u, b_i^d, b_j^u, b_j^d$, el flujo creado en vértices fuente o consumido en vértices sumidero x_i y el flujo por una arista y_j. Las ecuaciones resultantes son:

$$\min \sum_{i \in V} w_i x_i + \sum_{j \in E} w_j y_j$$

$$x_i = \sum_{k:Out(i)} y_k, \quad i \in F$$

$$\sum_{k:In(i)} y_k = x_i, \quad i \in S$$

$$\sum_{k \in :n(i)} y_k = \sum_{k:Out(i)} y_k, \quad i \in V - F \cup S$$

$$b_i^d \leq x_i \leq b_i^u, \qquad i \in F \cup S$$

$$b_j^d \leq y_j \leq b_j^u, \qquad j \in E$$

$$b_i^u \leq \sum_{k:In(i)} y_k \leq b_i^u, \qquad i \in V - F \cup S$$

Tal como como lo hemos enunciado tenemos un *Problema de Programación Lineal*.

Problema de la Mochila

Como ejemplo tomemos el problema de la *Mochila* que iremos resolviendo de varias formas. El problema de la *Mochila* 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 $ob_i = (w_i, v_i)$ dónde w_i, v_i son, respectivamente su peso y su valor unitario. Además la mochila tiene una capacidad C. El problema busca ubicar en la mochila el máximo número unidades de cada objeto 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:

$$\max \sum_{i=0}^{n-1} x_i v_i$$

$$\sum_{i=0}^{n-1} x_i w_i \leq C$$

$$x_i \leq m_i, \quad i \in [0, n-1]$$

$$int \quad x_i, \quad i \in [0, n-1]$$

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 tomarían valores reales.

Problema de la Asignación

En este problema tenemos una lista de agentes L y una lista de tareas T ambas del mismo tamaño n. El coste de que el agente i realice la tarea j sea c_{ij}. Se pretende asignar a cada agente una tarea y sólo una de tal forma que se ejecuten todas las tareas con el coste mínimo. Asumimos las variables binarias x_{ij} toman valor 1 si el agente i ejecuta la tarea j y cero si no la ejecuta. El problema puede ser planteado de la forma:

$$\min \sum_{i=0,j=0}^{n-1,n-1} x_{ij}\, c_{ij}$$

$$\sum_{j=0}^{n-1} x_{ij} = 1, \quad i \in [0, n-1]$$

$$\sum_{i=0}^{n-1} x_{ij} = 1, \quad j \in [0, n-1]$$

$$bin\, x_{ij}, \quad i \in [0, n-1], \; j \in [0, n-1]$$

Ahora las variables son binarias, toman valores cero y uno y, de nuevo, tenemos un Problema de Programación Lineal Entera. El primer conjunto de restricciones indica que cada agente i tiene que realizar una tarea y sólo una. El segundo conjunto de restricciones indica que cada la tarea j tiene que ser realizada por un agente y sólo uno.

Inversión de capital

El problema se puede encunciar de la siguiente forma: Supongamos que deseamos invertir una cantidad total T. Tenemos identificadas n oportunidades de inversión. Cada oportunidad de inversión requiere de una cantidad c_i y se espera un resultado con valor actual v_i con $i \in [0, n)$. ¿Que las inversiones debemos realizar con el fin de maximizar el valor actual total?

El problema anterior, de una forma abstracta, es similar al Problema de la Mochila visto anteriormente. Ahora vamos a considerar algunas restricciones adicionales que podríamos querer añadir. Por ejemplo, consideremos limitaciones del tipo siguiente:

1. Sólo podemos hacer 2 inversiones como máximo.
2. Si se hace la inversión r la s también se debe hacer.
3. Si se hace la inversión u la v no se puede hacer y viceversa.

Asumiendo que las variables binarias x_i toman valor 1 si se realiza la inversión i y cero en caso contrario podemos modlear el problema como:

$$\max \sum_{i=0}^{n-1} x_i v_i$$
$$\sum_{i=0}^{n-1} x_i c_i \leq T$$
$$\sum_{i=0}^{n-1} x_i \leq 2$$
$$x_r - x_s \leq 0$$
$$x_u + x_v \leq 1$$
$$bin \;\; i, \;\; i \in [0, n)$$

La restricción $x_r - x_s \leq 0$, asumiendo que las variables toman valores binarios tiene como soluciones posibles para (x_r, x_s) los pares $\{(0,0), (0,1), (1,1)\}$ y por lo tanto si $x_r = 1$ implica que $x_s = 1$.

La restricción $x_u + x_v \leq 1$, asumiendo que las variables toman valores binarios tiene como soluciones posibles para (x_r, x_s) los pares $\{(0,0), (0,1), (1,0)\}$ y por lo tanto se escoge una de las dos inversiones o ninguna pero no las dos a la vez.

Recubrimiento de conjuntos (Set Covering)

Dado un conjunto de elementos U de elementos e_i, $i \in [0, n-1)$, (llamado el universo) y un conjunto S de m conjuntos s_j, cuya unión es igual al universo, y un peso $w_i \geq 0$ asociado a cada conjunto, el problema de cobertura conjunto es identificar el subconjunto más pequeño de S cuya unión es igual al universo U. Por ejemplo, considere si el universo es $U = \{1,2,3,4,5\}$ y el conjunto $S = \{\{1,2,3\},\{2,4\},\{3,4\},\{4,5\}\}$, todos ellos con peso 1, podemos comprobar que la unión de todos elo conjunto en S es U y que podemos cubrir la totalidad de los elementos con los conjuntos $\{\{1,2,3\},\{4,5\}\}$.

El problema puede ser modelado mendiante un Problema de Programación Lineal Entera considerando las variables binarias x_j, $j \in [0, m)$ que tomarán el valor 1 si el conjunto s_j es escogido.

$$\min \sum_{j=0}^{m-1} w_j x_j$$
$$(\sum_{j:\varphi(i)} x_j) \geq 1, \quad i \in [0, n)$$
$$bin \; x_j, \quad j \in [0, m)$$

Siendo $\varphi(i) = \{j : e_i \in s_j\}$, es decir el conjunto de índices de conjuntos que incluyen al elemento e_i. Hay una restricción por cada elemento. Esto garantiza que ese elemento estará en alguno de los subconjuntos elegidos.

El Problema de es equivalente al. El problema conocido como *Hitting Set* parte de un grafo bipartito con vértices de tipos $V1$, $V2$ y se pretende encontrar un subconjunto mínimo de los

vértices en $V1$ tal que incluyan a todos los $V2$ entre sus vecinos. El problema es equivalente al del Recubrimiento de Conjuntos (*Set Cover*) anterior. Para ello escojamos como vértices en $V1$ los conjuntos en S, como vértices en $V2$ los elementos en U y añadiendo unan arista de un vértices en $V1$ a otro en $V2$ si el correspondiente conjunto incluye al elemento.

El *Problema del Recubrimiento de Vértices* (*Vertex Cover*) de un grafo consiste en buscar un subconjunto mínimo de vértices tal que cada arista del conjunto es incidente al menos a uno de los vértices escogidos. Construyendo un nuevo grafo con vértices $V1$ que asociados a los vértices del grafo original y vértices $V2$ a sus aristas e incluyendo en el nuevo grafo una arista por cada incidencia de arista y vértice en el grafo original vemos que este problema es equivalente al *Hitting Set* visto antes y por lo tanto también equivalente al *Set Cover*. En concreto siendo E el conjunto de aristas, considerando las variables binarias $x_j,\ j \in [0, m)$ que tomarán el valor 1 si el conjunto v_j es escogido, el Problema del *Vetex Cover* puede ser formulado como:

$$\min \sum_{j=0}^{m-1} w_j x_j$$
$$x_u + x_v \geq 1, \quad (u, v) \in E$$
$$bin\, x_j, \quad j \in [0, m)$$

Donde por $(u, v) \in E$ queremos representar el cojunto de aristas del grafo y para cada arista su respectivos extremos.

El problema del *Recubrimiento Mínimo de Aristas* consiste en encontrar un subconjunto mínimo de aristas que tal que cada vértice del grafo sea incidente con alguna de las escogidas. Este problema, como podemos ver, tabién puede ser considerado como un caso particular de *Set Cover*.

Problema de las estaciones de bomberos

Para ilustrar este modelo, consideremos el siguiente problema de localización: Una ciudad está considerando la ubicación de sus estaciones de bomberos. La ciudad se compone de n barrios. Cada barrio es vecino de otros barrios y la relación de vecindad se puede representar mediante un grafo no dirigido cuyos vértices representan los barrios y existe una arista entre dos barrios si son vecinos. Una estación de bomberos se puede colocar en cualquier barrio y es capaz de gestionar los incendios, tanto para de barrio y como de cualquier barrio vecino. El objetivo es minimizar el número de estaciones de bomberos.

Este problema podemos considerarlo como un ejemplo de *Set Cover* considerando los barrios como elementos del conjunto U y cada conjunto vecinos de un vértice como uno de los conjuntos. Si los índices de los vecinos de v_j los representamos por $N(j)$ y las variables binarias $x_j,\ j \in [0, m)$, que tomarán el valor 1 si el vértice v_j es escogido, el problema de las estaciones de bomberos puede ser formulado como:

$$\min \sum_{j=0}^{m-1} x_j$$

$$(\sum_{j:N(i)} x_j) \geq 1, \quad i \in [0, m)$$

$$bin\, x_j, \quad j \in [0, m)$$

Problema de la aerolínea (emparejamiento de tripulaciones)

El problema parte de un grafo dirigido, dónde cada vértice representa un aeropuerto y arista un vuelo que está marcado con la hora de salida y llegada. El objetivo que se busca es asignar tripulaciones a los vuelos con el objetivo de minimizar una función de coste dada.

Definimos un emparejamiento de vuelos como una serie de vuelos que podrían ser atendidos por una sola tripulación. Esto significa que la hora de salida de un vuelo de la serie de un aeropuerto debe ser posterior a la hora de llegada del vuelo anterior al mismo aeropuerto. Suponemos que cada tripulación debe dar servicio al menos a dos vuelos, de modo que cada emparejamiento contiene al menos dos vuelos y que podemos asignar varias tripulaciones a un vuelo, si es necesario, con el fin de transportar una tripulación a otro aeropuerto.

El costo de una tripulación es proporcional al intervalo de tiempo entre la primera hora de salida y la última vez llegada +5 horas.

Un enfoque del problema es construir en primer lugar un grafo de conexiones cuyos vértices son los vuelos y existen aristas entre vuelos que pueden formar un emparejamiento. El grafo resultante no tiene ciclos y se puede se construir en el tiempo $O(n)$. Cada emparejamiento buscado es un subcamino del grafo obtenido de longitud al menos dos y su coste es el intervalo de tiempo entre la primera hora de salida y la última llegada más 5 horas. Todos los emparejamientos posibles se pueden enumerar.

La solución del problema viene dada por un subconjunto de los emparejamientos posibles que cubra todos los vuelos y tenga coste mínimo y asignar a cada emparejamiento una tripulación. Un vuelo puede estar en varios emparejamientos. Si un vuelo está en varios emparejamientos elegidos tendrá asignadas varias tripulaciones una tendrá la responsabilidad del vuelo y las demás serán simplemente transportandas.

Este problema es un caso particular de recubrimiento de conjuntos visto arriba.

Al problema se le pueden añadir restricciones adicionales como la longitud máxima de un emparejamiento, un tiempo mínimo entre un vuelo y el siguiente, etc. Estas se pueden tener en cuenta filtrando la lista original de emparejamientos permitidos.

Problema de Empaquetado en Contenedores (Bin_Packing)

El problema se formula de la siguiente manera: Dado un conjunto de contenedores S de tamaño V y una lista de n elementos con tamaños a_i encontrar el mínimo número de contenedores necesarios para empaquetar todos los elementos.

Escogiendo las variables binarias x_{ij} que toman valor 1 si el elemento j es empaquetado en el bin i y las variables y_j tomando el valor 1 si el contenedor j es usado el problema puede ser formulado como:

$$\min B$$
$$B = \sum_{j=0}^{n-1} y_j$$
$$B \geq 1$$
$$\sum_{j=0}^{n-1} a_j x_{ij} \leq V y_i, \quad i = 0, \dots, n-1$$
$$\sum_{i=0}^{n-1} x_{ij} = 1, \quad j = 0, \dots, n-1$$
$$bin\, y_j, x_{ij}, \quad i, j = 0, \dots, n-1$$

Coloreado de Grafos

El *Problema de Coloreado de Grafos* consiste en buscar el mínimo número de colores tal que dando un color a cada vértice sea distinto el color asociado a dos vértices vecinos. Para modelarlo como un Problema de Programación Lineal Entera partimos de las variables binarias $y_k,\ k \in [0, n)$, que tomarán el valor 1 si el color k es usado y n es el número de vértices del grafo. Además introducimos las variables binarias x_{ik} que serán 1 si el vértice i se le asigna el color k. El modelo del problema es:

$$\min \sum_{k=0}^{n-1} y_k$$
$$\sum_{k=0}^{n-1} x_{ik} = 1, \quad i = 0, \dots, n-1$$
$$x_{ik} - y_k \leq 0, \quad i, k = 0, \dots, n-1$$
$$x_{ik} + x_{jk} \leq 1, \quad (i, j) \in E, k = 0, \dots, n-1$$
$$bin \;\; x_{ik}, \;\; y_k, \quad i, k = 0, \dots, n-1$$

Las restricción (1) garantiza que cada vértice tiene color y solo uno. Las restricción (2) el vértice i recibe de color k si este color se utiliza. La restricción (3) que si los vértices i, j son vecinos no pueden tener el mismo color.

La restricción $a - b \leq 0$, asumiendo que las variables toman valores binarios tiene como soluciones posibles para (a, b) los pares $\{(0,0), (0,1), (1,1)\}$ y por lo tanto si $a = 1$ implica que $b = 1$.

La restricción $a + b \leq 1$, asumiendo que las variables toman valores binarios tiene como soluciones posibles para (a, b) los pares $\{(0,0), (0,1), (1,0)\}$ y por lo tanto una de las dos variables toma valor 1 o ninguna de de ellas pero no las dos a la vez.

Problema del Viajante

EL Problema del Viajante (o TSP sus siglas en inglés responde a la siguiente pregunta: Dada una lista de ciudades y las distancias entre cada par de ellas, ¿cuál es la ruta más corta posible que visita cada ciudad exactamente una vez y regresa a la ciudad origen? Este problema se representa en general coomo un grafo no dirigido con pesos en las aristas y vértices que representan las ciudades.

Este problema se puede ser formular como uno de Progrmación Lineal Entera. Para ello sean las variables binarias x_{ij} , $i, j = 0, \dots, n$, siendo $n + 1$ el número de vértices del grafo, que tomarán el valor 1 si el camino incluye la arista que va de i a j y 0 en otro caso. Sea c_{ij} la distancia desde la ciudad i a la ciudad j. Sean, además, las variables enteras u_i. Entonces el modelo de programación lineal entera puede ser escrito como:

$$min \sum_{i=0}^{n} \sum_{j=0, j\neq i}^{n} c_{ij} x_{ij}$$

$$\sum_{i=0, i\neq j}^{n} x_{ij} = 1, \quad j = 0, \dots, n$$

$$\sum_{j=0, j\neq i}^{n} x_{ij} = 1, \quad i = 0, \dots, n$$

$$u_i - u_j + n x_{ij} \leq n - 1, \quad 1 \leq i \neq j \leq n$$

$$bin \ x_{ij}, \quad i, j = 0, \dots, n$$

$$int \ u_i, \quad i = 0, \dots, n$$

El problema se simplifica eliminando las variables x_{ij} para las que no exista una arista en el grafo.

El primer y segundo conjunto de igualdades aseguran que para cada ciudad en el camino elegido hay una y una soal ciudad anterior y otra posterior. Siempre que existan valores para las variables enteras $\mathrm{u_i}$ la última restricción obliga a que un solo camino cubra todas las ciudades y no dos o más caminos disjuntos.

Para justificar la última restricción mostramos que para cada recorrido que cubre todas las ciudades hay valores de las variables $\mathrm{u_i}$ que satisfacen las restricciones. En un recorrido solución etiquetemos la primera ciudad con 0. Sea $\mathrm{u_i} = \mathrm{t}$ si la ciudad i es visitada en el paso t ($\mathrm{i, t} = 1,2, \dots \mathrm{n}$). Por tanto $u_i - u_j \leq n - 1$ dado que u_i no puede ser mayor que n y u_j no puede ser menor que 1. La última restricción se satisface siempre que $x_{ij} = 0$ como

se puede comprobar. Si $x_{ij} = 1$ también se cumple ya que $u_i - u_j + nx_{ij} = t - (t+1) + n = n - 1$.

Por otra parte la última restricción asegura que las aristas elegidas forman una única secuencia de ciudades y no varias secuencias disjuntas. Para ello es sufiente mostrar que cada solución pasa por la ciudad 0. En efecto sumando las desigualdades $u_i - u_j + nx_{ij} \leq n - 1$ a lo largo de una subruta cerrada de k pasos que no pase por la ciudad 0 (por lo tanto $x_{ij} = 1$) resulta $nk \leq (n-1)k$ que es una contradicción.

Problema de las Reinas

El problema consiste en colocar n reinas en un tablero de ajedrez $n \times n$ de tal manera 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 $0..n-1$.

Escogiendo las variables binarias x_{ij} que toman valor 1 si ublicamos una reina en la casilla i, j el problema puede ser formulado como:

$$\sum_{i=0}^{n-1} x_{ij} = 1, \quad j = 0, \dots, n-1$$

$$\sum_{j=0}^{n-1} x_{ij} = 1, \quad i = 0, \dots, n-1$$

$$\sum_{(i,j):\varphi 1(d1)} x_{ij} \leq 1, \quad d1 = -(n-1), \dots, n-1$$

$$\sum_{(i,j):\varphi 2(d2)} x_{ij} \leq 1, \quad d2 = 0, \dots, 2n-2$$

$$bin\ x_{ij}, \quad i, j = 0, \dots, n-1$$

Dónde $\varphi 1(d1),\ \varphi 2(d2)$ son los conjuntos de casillas (pares de índices) asociados a la diagonal principal $d1$ y a la diagonal secundaria $d2$ respectivamente. Con más detalle:

$$\varphi 1(d1) = \{(i,j) | j - i = d1 \wedge i \in [0..n-1] \wedge j \in [0..n-1]\}$$
$$\varphi 2(d2) = \{(i,j) | j + i = d2 \wedge i \in [0..n-1] \wedge j \in [0..n-1]\}$$

En este problema no se trata de encontrar un mínimo. Se trata de encontrar una solución y puede haber muchas.

El problema puede ser modelado alternativamente usando las variables enteras $x_i, \quad i: [0, n-1]$ y las restricciones:

$$allDifferent(x_{0,}\dots,x_{n-1})$$
$$allDifferent(x_{0,}x_1-1,\dots,x_{n-1}-n-1)$$
$$allDifferent(x_{0,}x_1+1,\dots,x_{n-1}+n-1)$$
$$x_i<n,\quad i:[0,n-1]$$
$$int\ x_i,\quad i:[0,n-1]$$

Veremos más adelante como implementar la restricción $allDifferent(x_{0,}\dots,x_{n-1})$.

Problema del Sudoku

El problema consiste en rellenar con los enteros $[1..n]$ las casillas de un tablero $n\times n$ de tal manera cada fila, cada columna y cada subtabla todos los enteros $[1..n]$ y una sola vez cada uno de ellos. Siendo $n=mk$ con k el tamaño de cada subtabla y m el número de subtablas y m^2 el número de subtablas. Una reina amenaza a los cuadrados de la misma fila, de la misma columna y de las mismas diagonales. A cada casilla le podemos asignarlLas coordenadas i,j que toman valores en $0..n-1$, siendo la casilla $(0,0)$ la inferior izquierda. A cada subtablas le asociamos el índice t que toma valores en $0..n-1$ y se cumple $t=\left(\frac{j}{k}\right)k+i/k$ siendo / la división entera. Por ejemplo para $(i,j)=(2,4)$, $n=9,k=3$ tenemos $t=\left(\frac{4}{3}\right)3+\frac{2}{3}=3$. Concretemos el problema para $n=9$.

Sea $\varphi(t)=\{(i,j)|t=t=\left(\frac{j}{k}\right)k+i/k\}$ y escogiendo las variables binarias x_{ijv} que toman valor 1 si la casilla i,j toma el valor v el problema puede ser formulado como:

$$\sum_{i=0}^{n-1}x_{ijv}=1,\qquad j,v=0,\dots,n-1$$
$$\sum_{j=0}^{n-1}x_{ijv}=1,\qquad i,v=0,\dots,n-1$$
$$\sum_{(i,j):\varphi(t)}x_{ijv}=1,\qquad t,v=0,\dots,n-1$$
$$bin\ x_{ijv},\quad i,j,v=0,\dots,n-1$$

En este problema no se trata de encontrar un mínimo. Se trata de encontrar una solución y puede haber muchas.

Modelar el problema usando la restricción $allDifferent(x_{0,}\dots,x_{n-1})$ se deja como ejercicio.

Problema de Tareas y Procesadores

El problema se formula de la siguiente manera: Dado una lista de m tareas con duraciones d_j y un conjunto de n procesadores buscar la asignación de tareas a procesadores tal que el tiempo toal de ejecución sea mínimo.

Escogiendo las variables binarias x_{ij} que toman valor 1 si la tarea j es asignada al procesador i el problema puede ser formulado como:

$$\min \ T$$
$$\sum_{j=0}^{m-1} d_j x_{ij} \leq T, \quad i = 0, \dots, n-1$$
$$\sum_{i=0}^{n-1} x_{ij} = 1, \quad j = 0, \dots, m-1$$
$$bin \ x_{ij}, \quad i = 0, \dots, n-1, j = 0, \dots, m-1$$

Planificación

El problema se define por un conjunto tareas $j: \{1..n\}$, y un conjunto de medios (máquinas) $i: \{1..m\}$. Cada tarea j tiene que ser asignada a una máquina i dónde consume un tiempo de procesamiento p_{ij} y recursos a una tasa de c_{ij}. Cada tarea tiene una fecha de lanzamiento r_j y un tiempo de finalización d_j. Además cada tarea j asignada a la máquina i debe comenzar en un tiempo s_j tal que, cumpliendo las restricciones anteriores, la tasa de consumo de recursos en la máquina i nunca sea mayor que C_i en cualquier punto del tiempo. Para modelar el problema discretizamos el tiempo, definimos la variable $t: \{0..N\}$ y las variables binarias x_{ijt} que tomarán valor 1 si la tarea j comienza a ejecutarse en la máquina i en el tiempo t.

Este es un problema complejo con muchas variantes del cual sólo daremos aquí algunas ideas.

Como hemos dicho antes se debe cumplir una primera restricción que es:

$$s_j \geq r_j, \quad j: [1..n]$$

Una segunda restricción es que las tareas acaben antes de tiempo de finalización indicado. Pero esta restricción la vamos a considerar como una restricción blanda. Es decir vamos a quitarla como restricción y establecer como objetivo minimizar la demora total. Es decir la suma de los tiempos en que se ha sobrepasado el tiempo de finalización para cda tarea. Usaremos las variables $T_j \geq 0$ que deben cumplir:

$$T_j \geq s_j + p'_j - d_j, \quad j: 1..n$$

Dónde p'_j es el tiempo de ejecución de la tarea j asumiendo conocía la máquina en la que se ejecutará. Que podemos reescribir como:

$$T_j \geq \sum_{i:1..m} (t + p_{ij}) x_{ijt} - d_j, \quad j: 1..n, t: 1..N$$

Para comprender lo anterior debemos tener en cuenta que cada tarea j empieza en un único punto del tiempo t en la máquina i y por lo tanto la variable x_{ijt} toma el valor 1 sólo para esa combinación de valores paa una tarea dada. Por lo tanto:

$$s_j = \sum_{i:1..m} t x_{ijt}\,, \quad j:1..n, t:1..N$$
$$p'_j = \sum_{i:1..m} p_{ij} x_{ijt}\,, \quad j:1..n, t:1..N$$

Ahora la función objetivo es $g(x,s) = \sum_j T_j$.

Un segundo objetivo podría ser minimizar el número de tareas que acaban tarde. Para ello usamos las variables binarias L_j que deben ser 1 si la tarea j acaba tarde. Es decir si $s_j + p_{ij} > d_j$. Por lo tanto debemos incluir entre las restricciones la implicación $s_j + p_{ij} > d_j \rightarrow L_j = 1$. Esto se puede conseguir mediante:

$$NL_j \geq \sum_{i:1..m} (t + p_{ij}) x_{ijt} - d_j, \quad j:1..n, t:1..N$$

La suma de las variables L_j será el objetivo a minimizar. Es decir $g(x,s) = \sum_j L_j$.

Sumando sobre los subconjuntos apropiados de estas variables podemos establecer las restricciones del problema.

$$\min \sum_{j:1..n} T_j$$
$$T_j \geq \sum_{i:1..m} (t + p_{ij}) x_{ijt} - d_j, \quad j:1..n, t:1..N$$
$$\sum_{i=1}^{m} \sum_{t=1}^{N} x_{ijt} = 1, \quad j:1..n$$
$$\sum_{j:1..n} \sum_{s:T_{ijt}} c_{ij} x_{ijs} \leq C_i, \quad i:1..m, t:1..N$$
$$x_{ijt} = 0, \quad j:1..n, i:1..m, t:T$$
$$bin\ x_{ijt}, \quad j:1..n, i:1..m, t:1..N$$

Siendo $T_{ijt} = \{t':1..N \,|\, t - p_{ij} < t' \leq t\}$, $T = \{t':1..N \,|\, t' < r_j \vee t' > N - p_{ij}\}$. Como podemos ver T_{ijt} es la ventana de tiempo anterior a t en los cuales la tarea j puede haber empezado a ejecutarse en la máquina i. Y T en el conjunto que incluye los instantes de tiempo en los cuales la tarea no puede empezar, ya que son anteriores al tiempo de lanzamiento r_j, o que acabarían después del horizonte de tiempo especificado.

Como hemos dicho las variables T_j representan el tiempo en que la tarea j ha superado el establecido para finalizar d_j. La primera restricción establece eso. La segunda indica que una tarea empieza en un único instante del tiempo en una máquina dada. La tercera que los

recursos necesarios en la máquina i son menores que los disponibles en cada instante del tiempo. La cuarta que cada tarea no puede empezar antes de r_j o después de $N - p_{ij}$.

O para minimizar el número de tareas que acaban tarde sustituimos la función objetivo y la primera restricción por:

$$\min \sum_{j:1..n} L_j$$
$$NL_j \geq \sum_{i:1..m} (t + p_{ij}) x_{ijt} - d_j, \quad j: 1..n, t: 1..N$$

Y declaramos las variables L_j como binarias.

La máquina en la que se ejecuta una tarea j podemos asociarla a la variable y_j de la forma:

$$y_j = \sum_{i,t} i x_{ijt}$$

La restricción de que dos tareas q, k sean asignadas a la máquina i y exista una precedencia entre ambas puede modelarse mediante las restricciones:

$$y_q = y_k$$
$$s_q + p_{iq} \leq s_k, \quad i: [1..m]$$

2.2 Algunas técnicas útiles en los modelos de Programación Lineal Entera

Aquí recogemos algunas técnicas conocidas para construir modelos de Programación Lineal Entera.

Valor absoluto

El valor absoluto de una variable $|x|$ puede ser eliminado en un modelo de Programación Lineal Entera, con las nuevas variables y restricciones:

$$x = a - b$$
$$|x| = a + b$$

Eso es adecuado asumiendo que una de las dos variables (a, b), que tienen que ser ambas mayor o igual que cero, debe ser cero.

Para conseguirlo, y asumiendo que $L \leq x \leq U,\ L \leq 0 \leq U$, podemos escribir:

$$x = a - b$$
$$|x| = a + b$$
$$a \leq Uy$$
$$b \leq |L|(1 - y)$$
$$bin\ y$$

Podemos ver que si $y = 1$ resulta $a \leq U, b = 0$ y si $y = 0$ entonces $a = 0, b \leq |L|$

Variables semicontinuas

Son variables que cumplen:

$$x = 0 \ \vee a \leq x \leq b$$

Pueden ser modeladas como:

$$x \geq ay$$
$$x \leq by$$
$$bin\ y$$

Están disponibles directamente en entornos como *LPSolve*.

Special Ordered Sets

Restricciones de tipo k son un tipo especial de restricciones entre una lista de n variables que obligan a que justamente k varaibles contiguas de las lista sean distintas de cero. Un caso particular es para $k = 1$.

Están disponibles en en entornos como *LPSolve*.

Restricciones combinadas con el operador or

Sean las m restricciones como:

$$g_1(x) \geq 0$$
$$\dots$$
$$g_m(x) \geq 0$$

De las que queremos que se satisfagan algunas de ellas y liberar el resto. Esto lo podemos modelar con las siguientes restricciones ampliadas:

$$g_1(x) + L_1(1 - y_1) \geq 0$$
$$\dots$$
$$g_m(x) + L_m(1 - y_m) \geq 0$$
$$y_1 + \dots + y_m \geq k$$
$$bin\ \ y_i, \quad i: [1, m]$$

Dónde L_i es un valor positivo tal que -L_i es una cota inferior de la expresión $g_i(x)$ en el contexto de todas las demás restricciones, es decir que $g_i(x) \geq -L_i$ siempre es válido. Las variables y_i son variables binarias que cuando toman valor 0 liberan la restricción correspondiente y cuando toman valor 1 obligan a que se cumpla. Estableciendo restricciones adicional sobre la suma de las y_i podemos indicar cuantas restricciones deben satisfacerse o cuantas pueden quedar liberadas. Con la última restricción indicamos que se deben satisfacer al menos k de ellas.

Claramente si asociamos la misma variable binaria a varias restricciones éstas serán liberadas u obligadas a que se cumplan a la vez.

Podemos, además, establecer relaciones lógicas entre las restricciones que se satisfacen. La restricción adicional $y_r - y_s \leq 0$, como vimos más arriba, hace que si $y_r = 1$ entonces $y_s = 1$. Es decir que si se cumple $g_r(x) \geq 0$ también se debe cumplir $g_s(x) \geq 0$ y por lo tanto hay una relación de implicación entre ambas. Si junto a la anterior añadimos $y_s - y_r \leq 0$ entonces la implicación se convierte en sí y sólo si. Es decir $g_r(x) \geq 0$, $g_s(x) \geq 0$ deben ser ambas verdaderas o estar liberadas.

La restricción de doble implicación anterior puede ser usada para modelar variables indicadoras que son variables binarias que toman el valor 1 si y sólo si alguna otra restricción se satisface. Variables indicadoras, asociadas a la restricción $g_r(x) \geq 0$, pueden modelararse con las ideas anteriores haciendo que $g_s(x) \geq 0$ sea de la forma $u - 1 \geq 0$ y u binaria.

La restricción adicional $y_u + y_v \leq 1$ escoge una de las dos o ninguna pero no las dos a la vez.

Modelado del producto de variables

En algunos casos es interesante usar el producto de dos varaibles en los modelos que deber estar formados por restricciones lineales. Veamos como hacerlo.

Si tenemos las variables binarias x_1, x_2 su producto, $y = x_1 x_2$, puede ser sustituido por las restricciones:

$$\begin{gathered} y \leq x_1 \\ y \leq x_2 \\ y \geq x_1 + x_2 - 1 \\ bin\ y \end{gathered}$$

Si tenemos la variable binaria x_1 y la real x_2 y asumiendo que $0 \leq x_2 \leq u$ entonces su producto, $y = x_1 x_2$, puede ser sustituido por las restricciones:

$$\begin{gathered} y \leq u x_1 \\ y \leq x_2 \\ y \geq x_2 - u(1 - x_1) \end{gathered}$$

Si mabas varaibles son reales, el producto también se puede siguiendo las ideas anteriroes pero no lo vamos a ver aquí.

Modelo de la restricción *allDifferent* y permutaciones de un conjunto

Sean las variables i, $i: [1..n]$ que toma valores enteros en un conjunto finito de enteros φ tal que $|\varphi| = m \geq n$ y queremos modelar la restricción:

$$allDifferent(x_{1,} \dots, x_n, \varphi) \equiv x_1 \neq x_2 \neq \cdots \neq x_n$$

Para ello usamos las variables binarias y_{iv}, $i: [i..n]$, $v: \varphi$ que toma el valor 1 si la variable x_i toma el valor v. Las restricciones equivalentes son:

$$\sum_{i=1}^{n} y_{iv} \leq 1, \quad v:\varphi$$
$$\sum_{v:\varphi} y_{iv} = 1, \quad i = 1 \dots n$$
$$x_i = \sum_{v:\varphi} v y_{iv}, \quad i = 1 \dots n$$

La primera restricción indica que para cada valor puede haber una variable como máximo que lo tenga. Si $m = n$ entonces eta restrcción se convierte en una igualdad. La segunda restricción indica que cada varaible tiene un solo valor. La tercera indica que cada varaible tiene el valor entero adecuado. Un caso concreto es cuando $\varphi = \{z | 1 \leq z \leq n\}$.

Podemos ver que cada uno de los valores posibles de la lista de variables $(x_1, x_2, \dots, x_n)$ es una permutación de un subconjunto de φ o de φ completo si $m = n$.

La tercera restricción también nos da pistas para modelar una variable entera que toma solamente los valores en un conjunto φ. Para ello es suficiente con usar variables binarias y_v, $v:\varphi$ que toma el valor 1 si la variable x toma el valor v. La restricciones asociadas son:

$$\sum_{v:\varphi} y_v = 1$$
$$x = \sum_{v:\varphi} v y_v$$

2.3 Complejidad de los problemas de Programación Lineal Entera

La complejidad de los problemas de *Programación Lineal* y los de *Programación Lineal Entera* es muy diferente.

Los problemas de *Programación Lineal* (PL) suelen resolverse por el algoritmo del Simplex que, aunque no sea teóricamente polinómico en el caso peor, en la práctica se comporta como polinómico en la mayoría de los casos de interés.

Si a las soluciones de un problema lineal les exigimos que sean sean enteras, nos encontramos con un problema de *Programación Lineal Entera* (PLE). Este problema es *NP-duro*, concepto que no abordaremos aquí, que indica que su complejidad es más alta que la polinomial y por lo tanto hace imposible la obtención de soluciones exactas para problemas de tamaños grandes. Los algoritmos que se utilizan para resolver este problema suelen comenzar relajando las restricciones del problema para convertirlo en un Problema de Programción Lineal y posteriormente utilizar técnicas de Ramifica y Poda para encontrar la solución entera exacta.

2.4 Programación Lineal Entera. Detalles de Implementación

Para resolver el problema disponemos de un algoritmo para buscar las soluciones:

- 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 AlgoritmoPLI createPLI(String fichero) {
	return AlgoritmoPLI.create(fichero);
}
```

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.

Por lo tanto para resolver un problema debemos cosntruir el fichero que lo describe en primer lugar. Para problemas de tamaño grande es conveniente generar el fichero automáticamente. El código para generar el fichero que resuelve el problema de las Reinas puede encontrarse aquí. El programa principal para este problema puede encontrarse aquí.

Para hacer más fácil el código el algoritmo AlgoritmoPLI proporciona algunos métodos para generar el nombre de variables indexadas, sumatorios de ese tipo de variables, etc. Algunos métodos son:

```
AlgoritmoPLI  create(String fichero);
String getName(int i);
List<String> getNames();
Integer getNumVar();
Double getObjetivo();
Double[] getSolucion();
String getVariable(String name, int i, int j);
String getVariable(java.lang.String name, int i, int j, int k);
String intVariables(java.lang.String name, int i1, int i2);
```

3. Algoritmos aleatorios y optimización

3.1 Introducción

La técnica de *Programación Lineal Entera* es muy versátil y puede ser usada para modelar y resolver un amplio abanico de problemas de optimización. Pero tiene tiene algunos inconvenientes. El primero es que los modelos deben estar formulados mediante restricciones lineales combinadas con el operador *and* y por el operador *or*. Los problemas que requieren restricciones no lineales no pueden ser modelos mediante esta técnica. Existen metodologías

más potentes que pueden tener en cuenta esas restricciones no lineales pero no las veremos en este curso. El segundo es que los algoritmos que resuelven los modelos de Programación Lineal entera si aparecen muchos operadores or explícita o implícitamente tienen una complejidad mayor que polinomial. Esto los hace intratatables y por lo tanto encontrar una solución exacta al problema planteado es inabordable en la mayoría de los casos.

El enfoque para resolver este problema es buscar soluciones aproximadas y renunciar de forma general a encontrar la solución exacta. Este enfoque tiene dos vías. La primera es relajar el problema de Programación Lineal Entera a otro de Programación Lineal eliminando el carácter entero o binario de las variables y declarándolas todas reales. El Problema de Programación Lineal se puede resolver mediante el algoritmo del Simplex que tiene complejidad polinomial y es muy eficiente para la inmesa mayoría de problemas. Pero la solución al problema relajado es una aproximación que después puede ser mejorada. Una manera de mejorar la solución es la t´ncia aleatoria conocida como Búsqueda Tabú

La segunda vía, que puede ser complementaria a la anterior, es usar directamente algoritmos aleatorios para buscar una solución aproximada del problema de optimización. Eso es lo que vamos a ver ahora concretándonos en lso *Algoritmos Genéticos* y la técnica de *Simulated Annealing.*

Veamos, en primer lugar, como caracterizar un problema de optimización y como transformarlo a otro dónde las restricciones se pasan a la función objetivo.

3.2 Problemas de optimización

Los problemas que 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 en el conjunto de restricciones:

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

Si las restricciones anteriores las denominamos Ω encontrar una solución que cumpla las restricciones es equivalente al problema de minimización

$$\min_{x \in \Omega} ? = \min_{a \leq x \leq b} T(x) = \min_{a \leq x \leq b} \sum_{i=1}^{r} (c(g_i(x)))^2 + \sum_{j=1}^{s} (h_j(x))^2$$

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:

$$(\min_{x \in \Omega} f_1(x), \min_{x \in \Omega} f_2(x), \dots, \max_{x \in \Omega} f_{m-1}(x), \max_{x \in \Omega} f_m(x))$$

Que cambinado los signos adecaudamente se puede transformar en:

$$\min_{x \in \Omega}(f_1(x), f_2(x) \dots, -f_{m-1}(x), -f_m(x)) = \min_{x \in \Omega}(g_1(x), g_2(x) \dots, g_{m-1}(x), g_m(x))$$

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

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

Donde el valor relativo de ω_i indicará la prioridad del objetivo correspondiente.

Las restricciones Ω de un problema pueden ser introducidas en la función objetivo introduciendo un primer objetivo de minimización más prioritario que los demás de la forma:

$$(\min_{x\in\Omega} f_1(x), \dots, \max_{x\in\Omega} f_m(x)) = (\min_x T(x), \min_x f_1(x), \dots, \max_x f_m(x))$$

Algunas equivalencias son:

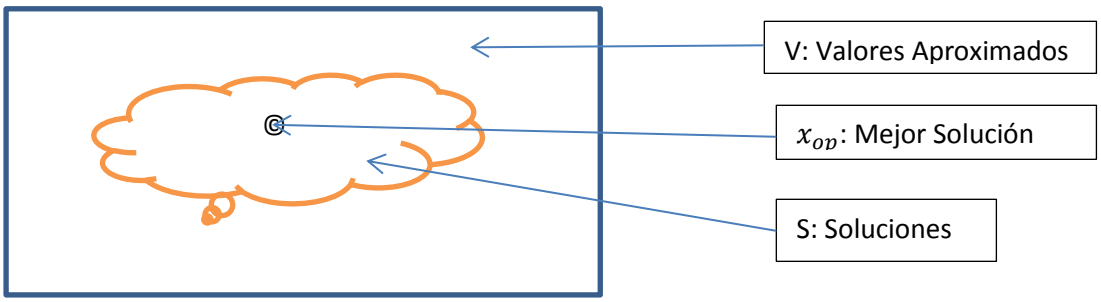
$$\begin{array}{l} \min\limits_x f(x) \\ g_i(x) \le 0,\ \ i = 1, \dots, r \\ h_j(x) = 0,\ \ j = 1, \dots, s \\ a \le x \le b \end{array} \equiv \min_x f(x) + R\Big(\sum_{i=1}^{r}(c(g_i(x)))^2 + \sum_{j=1}^{s}(h_j(x))^2\Big) \quad a \le x \le b$$

$$\begin{array}{l} \max\limits_x f(x) \\ g_i(x) \le 0,\ \ i = 1, \dots, r \\ h_j(x) = 0,\ \ j = 1, \dots, s \\ a \le x \le b \end{array} \equiv \min_x -f(x) + R\Big(\sum_{i=1}^{r}(c(g_i(x)))^2 + \sum_{j=1}^{s}(h_j(x))^2\Big) \quad a \le x \le b$$

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

3.3 Algoritmos probabilísticos

Los algoritmos genéticos (GA), simulated annealing (SA), y la búsqueda tabú (TS) son algoritmos iterativos generales de optimización combinatoria. Son algoritmos probabilístico cuyo diseño está inspirado en los mecanismos evolutivos que se encuentran en la naturaleza. Estos algoritmos tienen muchas similitudes, pero también poseen características distintivas, principalmente en sus estrategias para buscar en el espacio de estado de soluciones.



Los algortimos anteriores están orientados a resolver problemas de optimización. Un problema de optimización, como hemos dicho antes, pretende buscar la solución x_{op} que minimiza una función objetivo $g(x)$ que toma valores en el conjunto de soluciones S que cumplen las restricciones $\Omega(x)$. Así tenemos:

$$S = \{x: T | \Omega(x)\}$$
$$x_{op} = \min_{x:S} g(x)$$

Dónde T es el tipo de las soluciones del problema y, aunque hemos puesto *min*, podríamos haber considerado *max* en la misma forma. Al conjunto S se le suele llamar espacio de soluciones del problema. En los algoritmos que vamos a considerar introducimos un espacio de valores V y una *función de decodificación* $d(v)$ que toma valor en V y devuelve otro en T. El espacio V se escoje para que su valores se puedan generar fácilmente aplicando pertubaciones aleatorias a uno de ellos v_0 o mediante mezclas aleatorias de pares de ellos (v_0, v_1). Pasar del tipo T al V lo llamamos *codificar* y la revés *decodificar*. De forma general a las perturbaciones sobre un valor los llamaremos *operadores de mutación* y a los segundos *operadores de cruce*. En los algoritmos genéticos los valores del espacio de valores se le suele llamr cromosomas. Usaremos los mismos términos en los tres algoritmos probabilísticos que vamos a considerar.

Asociada al conjunto de restricciones $\Omega(x)$ asumimos la existencia de una función $R(x)$ que toma valores reales positivos y que cumple $\Omega(x) \rightarrow R(x) = 0$. Es decir $R(x) = 0$ si se cumplene las restricciones. Antes vimos la forma de obtener esa función a partir de las restricciones.

Por último definiremos la *función de fitness* como $f(v) = g\big(d(v)\big) + K\, R(d(v))$. Dónde estamos considerando un problema de minización y K una constante suficientemente grande. Esta función, como vemos, se define para cada valor (o cromosoma) que incluye las restricciones del problema y que pretendemos minimizar.

4. Un catálogo de tipos de cromosomas

Hay una amplia gama de problemas combinatorios que pueden ser abordados partiendo de un catálogo de cromosomas cuyos operadores de cruce y mutación son conocidos y pueden ser reutilizados. Este catálogo es útil en los algoritmos genéticos y las ideas pueden ser usadas en simulated annealing que son las técnicas que veremos con más detalle. Veamso los tipos adecuados para ser usados en los *Algoritmos Genéticos*.

Partimos del tipo genérico para construir los espacios de valores que llamaremos *IChromosome<T>*:

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

- `T decode()`:Función de decodificación
- `double fitness()`:Función de fitness

4.1 BinaryChromosome

Representa una lista de valores binarios. Es un cromosoma básico a partir del cual se pueden construir otros más complejos. Los operadores de mutación son muy conocidos y consisten fundamentalmente en permutar el valor binario de una casilla escogida al azar.

Los operadores de cruce son también ampliamentre conocidos y pueden consultarse en la literatura.

```
public interface ProblemaAGBinario<S> extends ProblemaAG {
	Double fitnessFunction(IBinaryChromosome cr);
	int getDimensionDelChromosoma();
	S getSolucion(IBinaryChromosome cr)
}
```

Restricciones:

- n = getDimensionDelChromosoma();
- d = decode();
- s = d.size();
- d==n;
- d[i]==1 || d[i]==0, i:0..n-1;

Usos:

Este cromosoma es adecuado para modelar un amplio abanico de situaciones. Puede ser considerado el cromosoma básico a partir del cual construir otros. Con él podemos modelar todos los problemas de *Programación Lineal Entera* con variables binarias.

Implementación:

Una implementación puede encontrarse en la clase BinaryChromosome2.

Ejemplo 1: Problema de la Asignación

En este problema tenemos una lista de agentes L y una lista de tareas T ambas del mismo tamaño n. El coste de que el agente i realice la tarea j sea c_{ij}. Se pretende asignar a cada agente una tarea y sólo una de tal forma que se ejecuten todas las tareas con el coste mínimo.

Una primera versión del problema es hacer una implementación de la solución propuesta más arriba mediante la técnica de la *Programación Lineal Entera*. En la solución asumimos las variables binarias x_{ij} toman valor 1 si el agente i ejecuta la tarea j y cero si no la ejecuta. Decimos que hemos codificado el problema mediante las variables binarias x_{ij}. La solución en *Programación Lineal Entera* fue:

$$\min \sum_{i=0,j=0}^{n-1,n-1} x_{ij}\, c_{ij}$$

$$\sum_{j=0}^{n-1} x_{ij} = 1, \quad i \in [0, n-1]$$

$$\sum_{i=0}^{n-1} x_{ij} = 1, \quad j \in [0, n-1]$$

$$bin\, x_{ij}, \quad i \in [0, n-1], \; j \in [0, n-1]$$

El cromosoma *IBinaryChromosome* dispone de un vector d, el resultado de la decodificación, que debe contenedor todas la varaibles binarias en un total de n^2 que será el tamaño del cromosoma. Identificamos cada variable x_{ij} con una posición k en el vector d en la forma: $x_{ij} = d[k]$ si $k = ni + j$. La función de fitnes debe tener en cuenta las restricciones tal como hemos explicado arriba. Con todo ello la función de fitnes, teniendo en cuenta que los Algoritmos Genéticos buscan el maximizar, puede ser:

$$f = -\left(\sum_{i=0,j=0}^{n-1,n-1} x_{ij}\, c_{ij}\right) - K(\sum_{i=0}^{n-1}(\sum_{j=0}^{n-1} x_{ij}) - 1)\text{^}2 + \sum_{j=0}^{n-1}((\sum_{i=0}^{n-1} x_{ij}) - 1)\text{^}2)$$

Hay otra forma mejor de resolver este problema como veremos más adelante con un cromosoma de otro tipo. Escoger

La razón es que el porcentaje de valores de cromosomas que no válidos (que no cumplen las restricciones) es muy alto. En efecto el número total variables es n^2 y de valores posibles es 2^{n^2}. Dadas las restricciones de igualdad el número de variables libres es $n^2 - 2n$ y el total de valores válidos, que cumplen lass restricciones, es 2^{n^2-2n}. El cociente nos da la probabilidad de encontrar un valor válido al azar y es:

$$\frac{2^{n^2-2n}}{2^{n^2}} = \frac{2^{n^2}}{2^{2n}2^{n^2}} = \frac{1}{2^{2n}}$$

Este valor nos da una medida de la calidad de la codificación.

Ejemplo 2: Problema de las estaciones de bomberos

Una ciudad se compone de n barrios. Cada barrio es vecino de otros barrios y la relación de vecindad se puede representar mediante un grafo no dirigido cuyos vértices representan los barrios y existe una arista entre dos barrios si son vecinos. Queremos ubicar una estación de bomberos en algunos barrios con la restricción que en cada barrio o en uno de sus vecinos haya una estación de bomberos. El objetivo es minimizar el número de estaciones de bomberos. Si las varaibles binarias x_j indican si un barrio tendrá estación de bomberos o no y $N(j)$ los vecinos del barrio j, el problema de Programación Lineal Entera era:

$$\min \sum_{j=0}^{m-1} x_j$$
$$(\sum_{j:N(i)} x_j) \geq 1, \quad i \in [0,m)$$
$$bin\, x_j, \quad j \in [0,m)$$

Que podemos podemos reescribir mediante la función de fitness:

$$f = -\left(\sum_{j=0}^{m-1} x_j\right) - K \sum_{i=0}^{m-1} \varphi\left(\left(\sum_{j:N(i)} x_j\right) - 1\right)$$
$$\varphi(u) = \begin{cases} -u, & x < 0 \\ 0, & x \geq 0 \end{cases}$$

4.2 ListIntegerChromosome

Representa una lista de valores enteros. Es un cromosoma básico que se puede combinar con operadores de mutación de diferentes tipos. Los operadores de mutación son muy conocidos y consisten fundamentalmente en permutar el valor binario de una casilla escogida al azar dentro de un rango establecido o en intercambiar los valores de dos casillas escogidas al azar.

Los operadores de cruce son también ampliamentre conocidos y similares a los usados para el cromosoma binario anterior. Los detalles del problema:

```
public interface ListIntegerChromosome extends IChromosome<List<Integer>>  {
	ProblemaAGListInteger<?> getProblem();
}
```

```
public interface ProblemaAGListInteger<S> extends ProblemaAG {
	Double fitnessFunction(ListIntegerChromosome cr) ;
	int getDimension();
	MutationPolicy getMutationPolicy();
	List<Integer> getRandomList()
	S getSolucion(ListIntegerChromosome cr);
}
```

4.3 IndexChromosome

Representa una lista de valores enteros que pueden ser usados como índices a una lista de objetos dada.

```
public interface IndexChromosome extends IChromosome<List<Integer>>  {
	Integer getMax(int i);
```

```
        Integer getObjectsNumber();
        ProblemaAGListInteger<?> getProblem();
}
```

```
public interface ProblemaAGIndex<S> extends ProblemaAG {
        Double fitnessFunction(IndexChromosome cr) ;
        Integer getMax(int index);
        List<Integer> getNormalSequence();
        Integer getObjectsNumber();
        S getSolucion(IndexChromosome cr);
}
```

Restricciones y notación:

- `n = getObjectsNumber();`
- `r = getNormalSequence().size();`
- `d = decode();`
- `s = d.size();`
- `n<=s<=r;`
- `m(i) = getMax(i);`

La secuencia normal asociada al problema está formada por la concatenación de *n* sublistas *L(i)* . Cada *L(i)* está se compone de *getMax(i)* copias del entero *i,* con i en el rango *0..n-1*.

El valor decodificado es una lista con enteros

La documentación puede encontrarse en IndexChromosome y ProblemaAgIndex.

4.4 IndexSubListChromosome

Es un subtipo de IndexChromosome cuyos valores son un subconjunto de la secuencia normal sin importar el orden.

Añade las siguientes restricciones:

- `0<=d[i]<n, i:s-1;`
- `toMultiset(d).count(i) < m(i), i:0..n-1;`

Usos:

Es un cromosoma adecuado para resolver problemas cuya solución es un *Multiset* formado con elementos de un conjunto dado.

Implementación:

Una implementación puede encontrarse en IndexChromosomeSubList.

Ejemplo: Problema de la Mochila

Se parte de L, una lista de objetos de tamaño n, y m una lista de enteros del mismo tamaño dónde $m(i)$ indica el número de repeticiones posibles del objeto en la posición i. A su vez cada objeto ob_i de la lista es de la forma $ob_i = (w(i), v(i))$ dónde $w(i), v(i)$ son, respectivamente, su peso y su valor unitario. Además la mochila tiene una capacidad C. El problema busca ubicar en la mochila el máximo número unidades, siempre que no superen las máximas permitidas para cada tipo de objeto, que quepan en la mochila para que el valor de los mismos sea máximo.

Si d es el valor decodificado entonces la función de fitness es:

$$f = \sum_{i=0}^{s-1} v(d[i]) - K\varphi((\sum_{i=0}^{s-1} w(d[i]) - C)$$

$$\varphi(u) = \begin{cases} 0, & u \leq 0 \\ u, & u > 0 \end{cases}$$

Hemos de tener en cuenta que d es un vector cuyas casillas son índices a la lista L que se repiten un máximo indicado en el problema.

4.5 IndexRangeChromosome

Es un subtipo de IndexChromosome cuyos valores son listas de enteros con las siguientes restricciones:

- `s=n`
- `0<=d[i]<=m(i), i:n-1;`

Es decir la lista decodificada es de tamaño igual al número de objetos y cada casilla de la misma es un entero positivo o cero y menor o igual que el máximo permitido para ese tipo de objetos.

Usos:

Es un cromosoma adecuado para resolver problemas cuya solución es un *Multiset* formado con elementos de un conjunto dado u otros problemas en los que aparecen variables enteras en un rango.

Implementación:

Una implementación puede encontrarse en IndexChromosomeRange.

Ejemplo: Problema de la Mochila

Ya lo hemos explicado arriba. Ahora las casillas de la lista decodificada es el número de unidaes escogido de cada objeto. Si d es la lista decodificada entonces la función de fitness es:

$$f = \sum_{i=0}^{n-1} d[i]v(i) - K\varphi((\sum_{i=0}^{s-1} d[i]w(i)) - C)$$

$$\varphi(u) = \begin{cases} 0, & u \leq 0 \\ u, & u > 0 \end{cases}$$

Se puede conseguir una implementación equivalente usando el cromosoma *ListIntegerChromosome* anterior junto con un operador de mutación de perturbe una casilla al azar.

4.6 IndexPermutationRandonKeyChromosome

Es un subtipo de *IndexChromosome* cuyos valores son listas de enteros que son permutaciones de la secuencia normal. Cada entero es un índice a la lista de objetos proporcionada. Añade las siguientes restricciones:

- `s=r`
- `0<=d[i]<n, i:0..s-1;`

Usos:

Es un cromosoma adecuado para resolver problemas cuya solución es una permutación de un multiconjunto dado de objetos.

Implementación:

Una implementación puede encontrarse en IndexChromosomePermutationRandomKey.

Ejemplo: Problema de la Asignación

Como ya vimos en este problema tenemos una lista de agentes L y una lista de tareas T ambas del mismo tamaño n. El coste de que el agente i realice la tarea j sea c_{ij}. Se pretende asignar a cada agente una tarea y sólo una de tal forma que se ejecuten todas las tareas con el coste mínimo.

Si asumimos que $d[i]$ es la tarea asignada al agente i la función de fitness se puede escribir como:

$$f = -\sum_{i=0}^{n-1} c_{id[i]}$$

Si comparamos esta solución con la que usaba el *IBinaryChromosome* podemos ver sus ventajas. Aquí todos los valores son válidos y la función de fitness y mucho más simple de escribir.

Se puede conseguir una implementación equivalente usando el cromosoma *ListIntegerChromosome* anterior junto con un operador de mutación de itercambie los valores de dos casillas al azar.

4.7 IndexPermutationSubListChromosome

Es un subtipo de *IndexChromosome* cuyos valores son listas de enteros que son subconjuntos de permutacionaciones de la secuencia normal. Cada entero es un índice a la lista de objetos proporcionada. Añade las siguientes restricciones:

- `s<=r`
- `0<=d[i]<n, i:0..s-1;`

Usos:

Es un cromosoma adecuado para resolver problemas cuya solución es una permutación de un subconjunto de un multiconjunto dado de objetos.

Implementación:

Una implementación puede encontrarse en PermutationIndexSubListChromosome

Ejemplos: El problema de los anuncios simplificado.

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. La secuencia de anuncios del año durará T segundos como máximo. Hay una lista L, de tamaño n, de anuncios que se ofertan para ser emitidos. Cada anuncio anuncio $a(i)$ tiene un tiempo de duración $t(i)$ y está dispuesto a pagar un precio $p(i) = \frac{b\ t(i)}{pos(i)} + c$. Dónde b, c son constantes y $pos(i)$ es la posición en la que se emitirá el anuncio si llega a ser emitido. Se quiere emitir el subconjunto de L cuyo tiempo total de emisión sea menor o igual que T y maximice el precio total de los anuncios.

La función de fitness es:

$$f = \sum_{i=0}^{s-1}(\frac{b\ t(d[i])}{i+1} + c) - K\varphi(\sum_{i=0}^{s-1} t(d[i]) - T)$$

$$\varphi(x) = \begin{cases} 0, & x \leq 0 \\ x, & x > 0 \end{cases}$$

4.8 RealChromosome

Los valores de estos cromosomas son listas de números reales en rangos especificados.

```
public interface IRealCromosome extends IChromosome<List<Double>>  {
	Integer getNum();
	Double getSup(int i);
	Double getInf(int i);
	ProblemaAGReal<?> getProblema();
}
```

Restricciones:

- s==getNum();
- getInf(i) <= d[i] <= getSup(i), i:0..s-1;

Usos:

Este cromosoma es adecuado para modelar funciones de varias variables reales del tipo $f(x_0, x_1, \dots, x_{n-1})$ de las que se quiere obtener el máximo o el mínimo en un dominio.

Implementación:

Una implementación y la documentación puede encontrase en IRealChromosome y RealListChromosome.

4.9 ExpressionChromosome

Los valores de este cromosoma son expresiones construidas con un conjunto de operadores, un conjunto de variables y otro constantes cuyos valores hay que determinar.

```
public interface IExpresssionChromosome<T> extends IChromosome<Exp<T>> {
	Integer getNumOperators();
	Integer getNumVariables();
	Integer getNumConstants();
	Exp<T> getExp();
	Exp<T> getOperator(int i);
	VariableExp<T> getVariable(int i);
	ConstantExp<t> getConstant(int i);
	ProblemaAGExpression<?,T> getProblem();
}
```

```
public interface ProblemaAGExpression<S,T> extends ProblemaAG {
	T convert(java.lang.Integer e) ;
	Double fitnessFunction(IExpressionChromosome<T> chromosome);
	Integer getMaxValueConstant();
	NAryExp<T> getNaryExp();
	Integer getNumOperators();
	Integer getNumVariables();
	Integer getNumConstants();
	List<Exp<T>> getOperators();
	S getSolucion(IExpressionChromosome<T> chromosome) ;
}
```

Usos:

Este cromosoma es adecuado para encontrar expresiones que cumplan con algunos requisitos.

Implementación:

La documentación e implementación se pueden encontrar en IExpressionChromosome, ProblemaAGExpression y ExpressionChromosome.

Ejemplo:

Dadas dos listas de valores Lx, Ly con valores para la variables x, y encontrar una función h tal que $y = h(x)$ que sea una aproximación a ellos.

La función de fitness es:

$$f = -\sum_{i=0}^{n-1} (e(Lx[i]) - Ly[i])^2$$

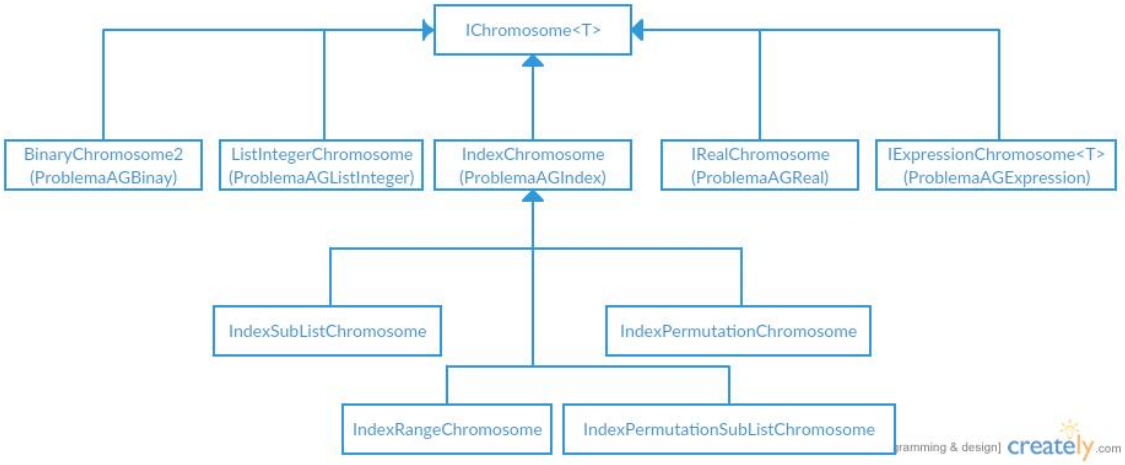
Dónde $e(x)$ es la expresión aociada al cromosoma evaluada en x. El código del ejemplo y un test se puede encontrar en TestExpresión.

4.10 Factoría de cromosomas

Una factoría para crear los distintos tipos de cromosomas está disponible en factoría de cromosomas.

El gráfico siguiente muestra los distintos tipos de comosomas y la relación entre los mismos. Las hojas del grafo son clases y las restantes interfaces. Hay site tipos distintos de cromosomas que denotan cada una de las clases del diagrama anterior. Los tipos son: *Binary, ListInteger, IndexSubList, IndexRange, IndexPermutation, IndexPermutationSubList, Real, Expression*.

Para cada tipo de cromosom es necesaria una información que se aporta en distintos interfaces que especializan *ProblemaAG*. Estos tipos son: *ProblemaAGBinary<S>, ProblemaAGExpression<S,T>, ProblemaAGIndex<S>, ProblemaAGListInteger<S>, ProblemaAGReal<S>*. En el diagrama se incluye entre paréntesis el tipo de problema asociado a cada tipo de cromosoma.



5. 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.

6. 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;
}
```

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.

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();
}
```

6.1 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 combinación de las anteriores

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

```
public static AlgoritmoAG createAG(ChromosomeType tipo, ProblemaAG p);
```

7. 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 optimizarla función objetivo. Aquí para reutilizar material de los Algoritmos Genéticos maximizaremos 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

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

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.

Operadores de mutación

Más arriba hemos diseñado un conjunto de cromosomas. Esas implementaciones la podemos usar directamente en los Algoritmos Genéticos. En *Simulated Annealing* podríamos usar los mismos cromosomas pero usando sólo los operadores de mutación y descartando los de cruce y selección. Aquí abordaremos la técnica de *Simulated Annealing* implementado cromosomas para cada caso concreto y dotándolos de operadore de mutación. Veamos algunos.

Los cromosomas binarios pueden dotarse de un operador del tipo $ob(c)$, con $0 \leq c < n$, que indica cambiar el contenido de la casilla c.

Los cromosomas lista de enteros (similares al *IntegerIndexChromosome*) 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.

En los cromosomas de permutación el operador de mutación más complejo. Se usa una lista h, del mismo tamaño, formada por valores reales comprendidos entre cero y uno. 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.

Otro operador posible en los cromosomas de permutación es $ol(a, c)$ con $0 \leq a < |l|,\ 0 \leq c < |l|,\ a \neq c$. Aplicar el operador significa permutar los valores de las casillas a, c.

Los cromosomas reales y sublista se implementan a partir de los demás y por lo tanto heredan sus operadores de mutación.

7.1 Busqueda Tabú

La solución inicial se obtiene redondeando las variables enteras alrededor sus valores óptimos en relajaciones LP. La búsqueda tabú comienza a modificar esta solución haciendo cambios exclusivamente en las variables enteras. Modificaciones de la solución se realizan en una estructura de vecindad simple: incrementar o decrementar en una unidad el valor de una variable entera. La vecindad una solución aproximada x está formada por aquellos valores x' que difieren en una unidad en un elemento x_i. Por lo tanto $x'_i = x_i + 1$ o $x'_i = x_i - 1$ y eel resto de las variables iguales.

En la búsqueda tabú los movimientos de una variable que se ha cambiado recientemente son prohibidos y son aceptados si conducen a la mejor a una mejor solución que la encontrada hasta el momento. La búsqueda se complementa con intensificación y diversificación. Intensificación permite a variables no prohibidas mejorar su valor mediante técnicas de ramifica y poda que veremos más adelante. La diversificación crea nuevas soluciones basadas en relajaciones, pero manteniendo una parte de la solución actual sin cambios.

Aquí no veremos esta técnica con mucho detalle.

8. Detalles de implementación de los Algoritmos de Simulated Annealing

El algortimo de Simulated Annealing podemos verlo como una versión simplificada de los algoritmos genéticos. Usaremos también los mismos tipos de cromosomas pero ahora sólo los operadores de mutación.

Los datos del problema los incluiremos en una clase que implemente el interface adecuado según el tipo de cromosoma. Posteriormente llamaremos al algoritmo con la factoría Algoritmos:

```
ProblemaReinasAG p = ProblemaReinasAG.create();
AlgoritmoSA ap = Algoritmos.createSA(ChromosomeType.IndexPermutation,p);
```

Fijados los tipos anteriores podemos implementar el esquema del *Algoritmo de Simulated Annealing.* Una versión simplificada de este algoritmo es:

```
public void ejecuta() {
	this.mejorSolucionEncontrada =
		ChromosomeFactory.randomChromosome(this.type);
	for (Integer n = 0; !parar && n < numeroDeIntentos; n++) {
		this.temperatura = temperaturaInicial;
		this.estado = ChromosomeFactory.randomChromosome(this.type);
		for (int numeroDeIteraciones = 0; !parar
			&& numeroDeIteraciones < numeroDeIteracionesPorIntento;
				numeroDeIteraciones++) {
			for (int s = 0; !parar && s <
				numeroDeIteracionesALaMismaTemperatura; s++) {
			   this.nextEstado = (IChromosome<?>)
			   this.mutationPolicy.mutate(this.estado.asChromosome());
			   double incr = nextEstado.fitness() - estado.fitness();
			   if (aceptaCambio(incr)) {
				estado = nextEstado;
				actualizaMejorValor();
			   }
			   parar = this.stopCond.isSatisfied(this.soluciones);
			}
			this.temperatura= nexTemperatura(numeroDeIteraciones);
		}
		soluciones.addChromosome(this.estado.asChromosome());
	     }
	}

	private void actualizaMejorValor() {
		if (estado.fitness() > mejorSolucionEncontrada.fitness()) {
			mejorSolucionEncontrada = estado;
		}
	}

	private double nexTemperatura(int numeroDeIteraciones) {
		return alfa*temperatura;
//		return temperaturaInicial/Math.log(2+3*numeroDeIteraciones);
	}

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

Una versión más completa puede encontrarse en el API.

9. Ejemplos de algoritmos genéticos

9.1 Problema de la Mochila

Se parte de L, una lista de objetos de tamaño n, y m una lista de enteros del mismo tamaño dónde $m(i)$ indica el número de repeticiones posibles del objeto en la posición i. A su vez cada objeto ob_i de la lista es de la forma $ob_i = (w(i), v(i))$ dónde $w(i), v(i)$ son, respectivamente, su peso y su valor unitario. Además la mochila tiene una capacidad C. El problema busca ubicar en la mochila el máximo número unidades, siempre que no superen las máximas permitidas para cada tipo de objeto, que quepan en la mochila para que el valor de los mismos sea máximo.

Si d es el valor decodificado usando un cromosoma tipo *IndexChromosomeSubList* entonces la función de fitness es:

$$f = \sum_{i=0}^{s-1} v(d[i]) - K\varphi((\sum_{i=0}^{s-1} w(d[i]) - C)$$

$$\varphi(u) = \begin{cases} 0, & u \leq 0 \\ u, & u > 0 \end{cases}$$

Hemos de tener en cuenta que d es un vector cuyas casillas son índices a la lista L que se repiten un máximo indicado en el problema.

Si d es la lista decodificada usando un cromosoma tipo *IndexChromosomeBinary* entonces la función de fitness es:

$$f = \sum_{i=0}^{n-1} d[i]v(i) - K\varphi((\sum_{i=0}^{s-1} d[i]w(i)) - C)$$

$$\varphi(u) = \begin{cases} 0, & u \leq 0 \\ u, & u > 0 \end{cases}$$

El código para resolver el problema puede encontrarse en ProblemaMochilaAG, que define el problema de la mochila y en las clases TestMochilaAGBinary que usa un cromosoma de tipo BinaryIndexChromosome y TestMochilaAGInteger que utiliza un cromosoma de tipo IntegerIndexChromosome.

9.2 Problema de las Reinas:

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 $(0, Nr - 1)$.

Este problema lo podemos modelar por un cromosoma de tipo *ListIntegerChomosome,* junto con un un operador de mutación que iontercambie dos casillas al azar, o un *IndexChromosomePermutationRandonKey.* La función de fitness es:

$$f = -(2n - |\{i: 0..n-1 \bullet d[i] - i\}| - |\{i: 0..n-1 \bullet d[i] + i\}|)$$

Para ello ebemos usar un cromosoma inciar con los valores $(0,1,2,\dots,n-1)$. Cada reina estará ocupando la casilla $(i, d[i])$ estará en las diagonales principal y secundaria $d[i] - i, d[i] + i$ respectivamente. Para que cada reina esté en una diagonal principal y otra secundaria diferentes, los conjuntos formados por esos valores tienen que tener cardinal n.

9.3 Problema de los Anuncios

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. La secuencia de anuncios del año durará T segundos como máximo. Hay una lista L, de tamaño n, de anuncios que se ofertan para ser emitidos. Cada anuncio anuncio $a(i)$ tiene un tiempo de duración $t(i)$ y está dispuesto a pagar un precio $p(i) = \frac{b\ t(i)}{pos(i)} + c$. Dónde b, c son constantes y $pos(i)$ es la posición en la que se emitirá el anuncio si llega a ser emitido. Se quiere emitir el subconjunto de L cuyo tiempo total de emisión sea menor o igual que T y maximice el precio total de los anuncios.

La función de fitness es:

$$f = \sum_{i=0}^{s-1}(\frac{b\ t(d[i])}{i+1} + c) - K\varphi(\sum_{i=0}^{s-1} t(d[i]) - T)$$

$$\varphi(x) = \begin{cases} 0, & x \leq 0 \\ x, & x > 0 \end{cases}$$

El problema puede ser ampliado para tener en cuenta otras restricciones como por ejemplo que existan anuncios

9.4 Problema de regresión

Dadas dos listas de valores Lx, Ly con valores para la variables x, y encontrar una función h tal que $y = h(x)$ que sea una aproximación a ellos.

La función de fitness es:

$$f = -\sum_{i=0}^{n-1}(e(Lx[i]) - Ly[i])^2$$

9.5 Problema de extremo de una función

Dada una función en un rango y con unas restricciones encontrar su mínimo o su máximo.

$$\max_{a \leq x \leq b,\ x \in \Omega} h(x)$$

La función de fitness es de la forma

$$f = h(d) - K * R(d)$$

Usando el cromosoma *RealChromosome* el valor decodificado es una lista de valores reales en el rango especificado. La función $R(d)$ mide la distancia del punto a la zona definida por las restricciones