**Resolución de Problemas:**

Deseamos definir cualquier tipo de problema de manera que se pueda resolver automáticamente, por tanto, necesitamos:

* Una representación común para todos los problemas.
* Algoritmos que usen alguna estrategia para resolver problemas definidos en esa representación común.

Elementos de un problema:

* Punto de partida (elementos que definen las características del problema).
* Objetivo que se quiere alcanzar (qué queremos obtener con la resolución).
* Acciones a nuestra disposición para resolver el problema (herramientas).
* Restricciones sobre el objetivo (características que debe tener la solución).
* Elementos que son relevantes en el problema definidos por el tipo de dominio.

Diferentes maneras de representación general de un problema:

* Espacio de estados: un problema se divide en un conjunto de pasos de resolución desde el inicio hasta el objetivo.
* Reducción a subproblemas: un problema se puede descomponer en una jerarquía de subproblemas.

Representaciones para problemas específicos: resolución de juegos, satisfacción de restricciones.

**Problema** -> elementos que intervienen + relaciones entre ellas

En cada instante de la resolución de un problema, esos elementos tendrán unas características y relaciones específicas.

**Estado** -> representación de los elementos que describen el problema en un momento.

* Estado Inicial -> punto de partida.
* Estado Final -> objetivo del problema.

**Operador** -> función de transformación sobre la representación de un estado que lo convierte en otro estado, por lo tanto, nos permite movernos entre los diferentes estados. También definen una relación de accesibilidad entre estados.

Representación de un operador:

* Condiciones de aplicabilidad (condiciones del estado para poder aplicarlo).
* Función de transformación (para conseguir un estado sucesor al actual).

**Espacio de estados** -> lo conforman los estados y su relación de accesibilidad. Representa todos los caminos que hay entre todos los estados posibles de un problema. La solución de nuestro problema está dentro de este espacio.

**Solución** -> secuencia de pasos que llevan del estado inicial al final (secuencia de operadores) o también el estado final.

Hay muchos tipos de solución, puede ser cualquiera, la mejor, todas.…

El coste de una solución es el gasto en recursos de la aplicación de los operadores a los estados.

*Descripción de un problema en Espacio de Estados*:

* Definir el conjunto de estados del problema.
* Especificar el estado inicial.
* Especificar el estado final o las condiciones que cumple.
* Especificar los operadores de cambio de estado (condiciones de aplicabilidad y función de transformación).
* Especificar el tipo de solución: secuencia de operadores o simplemente el estado final, una solución cualquiera o la mejor (definición de coste) …

*Búsqueda en el espacio de estados:*

La resolución de un problema con esta representación pasa por explorar el espacio de estados. Partiendo del estado inicial evaluando cada paso hasta encontrar un estado final. En el caso peor, exploraremos todos los posibles caminos entre el estado inicial del problema hasta llegar hasta el estado final.

*Estructura del espacio de estados:*

* Estructura de datos: árboles y grafos.
* Estados: nodos.
* Operadores: arcos entre nodos (dirigidos).
* Árboles: solo un camino lleva a un nodo.
* Grafos: varios caminos pueden llevar a un nodo.

El espacio de estados puede ser infinito y es necesaria una aproximación diferente para buscar y recorrer árboles y grafos (no podemos tener la estructura en memoria por la posible infinidad), por lo tanto, la estructura la construimos a medida que hacemos la búsqueda.

*Algoritmo básico:*

Seleccionar el primer estado como el estado actual

Mientras el estado actual no sea el estado final:

Generar y guardar sucesores del estado actual (expansión)

Escoger el siguiente estado entre los pendientes (selección)

La selección del siguiente nodo determinará el tipo de búsqueda (orden de selección o expansión).

Es necesario definir un orden entre los sucesores de un nodo (orden de generación).

Tipos de nodos:

* Nodos abiertos: estados generados, pero aún no visitados.
* Nodos cerrados: estados visitados y que ya se han expandido.

Puede ser necesario tener en cuenta los estados repetidos cuando exploremos un grafo (esto significa tener una estructura para los nodos cerrados).

Variando la estructura de nodos abiertos variamos el comportamiento del algoritmo (orden de visita de los nodos).

*Características de los algoritmos:*

* Completitud: si encuentra una solución.
* Complejidad temporal: el tiempo que tarda.
* Complejidad espacial: la memoria que gasta.
* Optimalidad: si encuentra la solución óptima.

*Tipos de algoritmos:*

1. Algoritmos de búsqueda ciega / búsqueda no informada:
   * No tienen en cuenta el coste de la solución en la búsqueda.
   * Su funcionamiento es sistemático, siguen un orden de visitas y generación de nodos establecido por la estructura del espacio de búsqueda.
   * Ejemplos: Anchura prioritaria, profundidad prioritaria, profundidad iterativa.
2. Algoritmos de búsqueda heurística y búsqueda local:
   * Utilizan una estimación del coste de la solución para guiar la búsqueda.
   * No siempre garantizan el óptimo, ni una solución.
   * Ejemplos: Hill-climbing, Branch and Bound, A\*, IDA\*.

**Búsqueda ciega / Búsqueda no informada:**

Los algoritmos de búsqueda no informada no dependen de información propia del problema a la hora de resolverlo. Por lo que son algoritmos generales y se pueden aplicar en cualquier circunstancia.

Se basan en la estructura del espacio de estados y determinan estrategias sistemáticas para su exploración. Siguen una estrategia fija a la hora de visitar los nodos que representan los estados del problema. Y también son algoritmos exhaustivos, pues pueden acabar recorriendo todos los nodos del problema para hallar la solución.

Por ser algoritmos exhaustivos y sistemáticos, su coste puede ser prohibitivo para la mayoría de los problemas reales, por lo tanto, solo será aplicables en problemas pequeños. La ventaja de estos algoritmos es que no nos hace falta obtener ningún conocimiento adicional sobre el problema, por lo que siempre son aplicables.

*Búsqueda en anchura prioritaria (BFS)*

Los nodos se visitan y generan por niveles, por lo que un nodo se visita solamente cuando todos sus predecesores y sus hermanos anteriores en orden de generación ya se han visitado. Estructura para los nodos abiertos -> cola (FIFO). Características:

* Completitud: siempre encuentra una solución.
* Complejidad temporal: exponencial respecto al factor de ramificación y la profundidad de la solución O(rp).
* Complejidad espacial: exponencial respecto al factor de ramificación y la profundidad de la solución O(rp).
* Optimalidad: la solución que se encuentra es óptima en número de niveles desde la raíz.

*Búsqueda en profundidad prioritaria (DFS)*

Los nodos se visitan y generan buscando los nodos a mayor profundidad y retrocediendo cuando no se encuentran nodos sucesores. La estructura para los nodos abiertos es una pila (LIFO). Para garantizar que el algoritmo acaba debe imponerse un límite en la profundidad de exploración. Características:

* Completitud: siempre encuentra una solución si dentro del límite de profundidad impuesto existe una solución.
* Complejidad temporal: exponencial respecto al factor de ramificación y la profundidad del límite de exploración O(rp).
* Complejidad espacial: en el caso de no controlar los nodos repetidos el coste es lineal respecto al factor de ramificación y el límite de profundidad O(rp). Si tratamos repetidos el ahorro en espacio es nulo por lo que el coste es igual que en anchura, exponencial respecto al factor de ramificación y la profundidad de la solución O(rp). Si la implementación es recursiva, el coste es O(p).
* Optimalidad: no se garantiza que se encuentre el óptimo.

*Búsqueda en profundidad iterativa*

Intenta combinar el comportamiento espacial del DFS con la optimalidad del BFS. El algoritmo consiste en realizar búsquedas en profundidad sucesivas con un nivel de profundidad máximo acotado y creciente en cada iteración. Así se consigue el comportamiento de BFS, pero sin su coste espacial, ya que la exploración es en profundidad, y además los nodos se regeneran a cada iteración. Además, esto permite evitar los casos en que DFS no acaba (existen ramas infinitas). En la 1ª iteración la profundidad máxima será 1 y este valor irá aumentando en sucesivas iteraciones hasta llegar a la solución. Para garantizar que el algoritmo acaba si no hay solución, se puede definir una cota máxima de profundidad en la exploración.

* Completitud: siempre encuentra una solución.
* Complejidad temporal: la misma que la búsqueda en anchura. El regenerar el árbol en cada iteración solo añade un factor constante a la función de coste O(rp).
* Complejidad espacial: en el caso de no controlar los nodos repetidos el coste es lineal respecto al factor de ramificación y el límite de profundidad O(rp). Si tratamos repetidos el ahorro en espacio es nulo por lo que el coste es igual que en anchura, exponencial respecto al factor de ramificación y la profundidad de la solución O(rp). Si la implementación es recursiva, el coste es O(p). (igual que la búsqueda en profundidad).
* Optimalidad: se garantiza que se encuentra el óptimo.

**Búsqueda heurística:**

Intenta hacer intervenir conocimiento sobre el problema que queremos resolver dentro del funcionamiento del algoritmo de búsqueda. El problema es que este conocimiento será particular para cada problema, por lo tanto, no será exportable a otros. Perderemos en generalidad, pero podremos ganar en eficiencia temporal.

Por lo general, tendremos un coste asociado a los operadores que nos permiten pasar de un estado a otro, por lo que ese coste tendrá un papel fundamental en el cálculo del coste del camino.

Supone la existencia de una función de evaluación que debe medir la distancia estimada al (a un) objetivo (h(n)). Esta función de evaluación se utiliza para guiar el proceso haciendo que en cada momento se selecciones el estado o las operaciones más prometedores. No siempre se garantiza encontrar una solución (si existe) ni tampoco garantiza encontrar la solución más próxima.

*Branch and Bound*

Ramificación y acotación:

* Generaliza BFS, DFS.
* Se guarda para cada estado el coste (hasta el momento) de llegar desde el estado inicial a dicho estado. Guarda el coste mínimo global hasta el momento.
* Deja de explorar una rama cuando su coste es mayor que el mínimo actual.
* Si el coste de los nodos es uniforme equivale a una búsqueda por niveles.

*Gredy Best First*

La estructura de abiertos es una cola de prioridad, la prioridad la marca la función de estimación (coste del camino que falta hasta la solución) y en cada iteración se escoge el nodo más cercano a la solución (el 1º de la cola), esto provoca que no se garantice la solución óptima. El estimador es importante.

**Búsqueda A\***

Dado que nuestro objetivo no es solo llegar lo más rápidamente a la solución, sino encontrar la de menor coste deberemos tener en cuenta el coste de todo el camino y no solo el camino por recorrer.

Coste de un arco entre dos nodos ni nj -> coste del operador que nos permite pasar de un nodo al otro, y lo denotaremos como c(ni,nj), este coste siempre será positivo.

Coste de un camino entre dos nodos ni nj -> suma de los costes de todos los arcos que llevan desde un nodo al otro y lo denotaremos como:

Diagrama, Esquemático

Descripción generada automáticamente

Coste del camino mínimo entre dos nodos ni nj -> camino de menor coste de entre los que llevan desde un nodo al otro y lo denotaremos como:

Texto, Esquemático

Descripción generada automáticamente

Si nj es un nodo terminal, llamaremos h\*(ni) a K(ni,nj), es decir, el coste del camino mínimo desde un estado cualquiera a un estado de solución.

Si ni es un nodo inicial, llamaremos g\*(nj) a K(ni,nj), es decir, el coste del camino mínimo desde el estado inicial a un estado cualquiera.

Esto nos permite definir el coste del camino mínimo que pasa por cualquier nodo como una combinación del coste del camino mínimo desde el nodo inicial al nodo más el coste del nodo hasta final: f\*(n) = g\*(n) + h\*(n)

Texto

Descripción generada automáticamente

Dado que desde un nodo específico n el valor de h\*(n) es desconocido, lo substituiremos por una función que nos lo aproximará, a esta función la denotaremos h(n) y será la función heurística. Esta función tendrá siempre un valor mayor o igual que cero. Denominaremos g(n) al coste del camino desde el nodo inicial al nodo n, evidentemente este coste es conocido ya que este camino lo hemos recorrido en nuestra exploración. De esta manera tendremos una estimación del coste del camino mínimo que pasa por cierto nodo: f(n) = g(n) + h(n)

* f es un valor estimado del coste total del camino que pasar por n.
* h (heurístico) es un valor estimado de lo que falta por llegar desde n al (a un) objetivo.
* g es un coste real (lo gastado por el camino más corto conocido hasta n).

Será este valor el que utilicemos para decidir en nuestro algoritmo cuál es el siguiente nodo por explorar de entre todos los nodos abiertos. La preferencia es siempre del nodo con menor f, en caso de empate, la preferencia es del nodo con menor h.

Con esta función podemos variar el comportamiento del algoritmo:

* Si todas las h(n) = 0, todo estará controlado por g (estamos en presencia de un algoritmo de Branch and Bound).
* Si todas las h(n) = 0 y además el coste de todos los arcos es 1, estaremos realizando una búsqueda en anchura. Si dicho coste fuera 0, la búsqueda sería aleatoria.
* Al ser h una estimación del verdadero coste h\*, cuanto más se aproxime h a h\* mayor será la tendencia a explorar en profundidad. Si h = h\*, entonces A\* converge directamente hacia el objetivo.

Si para todo n h(n) <= h\*(n) A\* encontrará (de haberlo) un camino óptimo.

*Tratamiento de repetidos*

Si es un repetido que está en la estructura de abiertos:

* Si su coste es menor, substituimos el coste por el nuevo, esto podrá variar su posición en la estructura de abiertos.
* Si su coste es igual o mayor, nos olvidamos del nodo.

Si es un repetido que está en la estructura de cerrados:

* Si su coste es menor, reabrimos el nodo insertándolo en la estructura de abiertos con el nuevo coste. No hacemos nada con sus sucesores, ya se abrirán si hace falta.
* Si su coste es igual o mayor, nos olvidamos del nodo.

*Admisibilidad*

El algoritmo A\* encontrará la solución óptima dependiendo del heurístico. Si el heurístico es admisible la optimalidad está asegurada.

Un heurístico es admisible si se cumple la siguiente propiedad:

Para todo n 0 <= h(n) <= h\*(n)

Por lo tanto, h(n) ha de ser un estimador optimista, nunca ha de sobreestimar h\*(n).

Diagrama

Descripción generada automáticamente

*Consistencia*

Podemos definir la propiedad de consistencia como una extensión de la propiedad de admisibilidad. Si tenemos el coste h\*(ni) y el coste h\*(nj) y el coste óptimo para ir de ni a nj, o sea K(ni,nj), se ha de cumplir la desigualdad triangular: h\*(ni) <= h\*(nj) + K(ni,nj).

La condición de consistencia exige pedir lo mismo al estimador h: h(ni) <= h(nj) + K(ni,nj).

Es decir, la diferencia de las estimaciones sea menor o igual que la distancia óptima entre nodos. Si esta propiedad se cumple esto quiere decir que h es un estimador uniforme de h\*. Es decir, la estimación de la distancia que calcula h disminuye de manera uniforme.

Si h es consistente, además se cumple que el valor de g(n) para cualquier nodo es g\*(n), por lo tanto, una vez llegamos a un nodo sabemos que hemos llegado a él por el camino óptimo desde el nodo inicial. Si esto es así, no es posible que nos encontremos el mismo nodo por un camión alternativo con un coste menor y esto hace que el tratamiento de nodos cerrados duplicados sea innecesario, lo que nos ahorra de tener que guardarlos.

Normalmente las funciones heurísticas admisibles suelen ser consistentes.

Diagrama

Descripción generada automáticamente con confianza media

* h(n) consistente => g(n) = k(s, n) (camino óptimo a n).
* Si g(n) = k(s, n), dado que h(n) siempre es constante => f(n) es mínima
* En este caso no es necesario tratar los nodos duplicados cerrados ya que los nodos expandidos ya no se podrán reexpandir (hemos llegado a ellos por el camino mínimo).

*Algoritmos más informados*

Dado un problema, existen tantos A\* para resolverlo como estimadores podamos definir.

Imagen que contiene Interfaz de usuario gráfica

Descripción generada automáticamente

* Si A\*1 es más informado que A\*1 entonces si el nodo n es expandido por A\*1 => n es expandido por A\*2 (pero no al revés).
* Eso quiere decir que A\*1 expande menor número de nodos que A\*2.

Esto nos haría pensar que siempre tenemos que escoger el heurístico más informado, pero esto no siempre es así. Evidentemente, cuanto más informado sea el heurístico, mayor será ese coste. Nos podemos imaginar, por ejemplo, que, si tenemos un heurístico que necesita por ejemplo 10 iteraciones para llegar a la solución, pero tiene un coste de 100 unidades de tiempo por iteración, tardará más en llegar a la solución que otro heurístico que necesite 100 iteraciones pero que solo necesite una unidad de tiempo por iteración. Esto nos hace pensar en que tenemos que encontrar un equilibrio entre el coste del cálculo de la función heurística y el número de expansiones que nos ahorramos al utilizarla. Es posible que una función heurística peor nos acabe dando mejor resultado. Todo esto es evidentemente dependiente del problema, incluso nos podemos encontrar con que una función no admisible nos permita hallar más rápidamente la solución, a pesar de que no nos garantice la optimalidad. Esto ha llevado a proponer variantes de A∗ donde se utilizan heurísticos cerca de la admisibilidad (e-admisibilidad) que garantizan una solución dentro de un rango del valor del coste óptimo.

*Óptimo con limitación de memoria*

El algoritmo A\* resuelve problemas en los que es necesario encontrar la mejor solución.

Su coste en espacio y tiempo en el caso medio es mejor que los algoritmos de búsqueda ciega si el heurístico es adecuado. Existen problemas en los que la dimensión del espacio de búsquedas no permite su solución con A\*. Además, los nodos a almacenar por A\* crecen exponencialmente si:

|h(n) – h\*(n)| >= log(h\*(n))

Existen algoritmos que permiten obtener el óptimo limitando la memoria usada, como por ejemplo, IDA\*, Best First Recursivo, Memory Bound A\* (MA\*)…

*Búsqueda IDA\**

Similar a ID (iteración de búsqueda en profundidad con un límite en la búsqueda). En ID el límite lo daba una cota máxima, en IDA\*, el límite lo da una cota máxima sobre el valor de la función f.

La búsqueda es una DFS normal y corriente, el heurístico sólo se utiliza para podar. Se empieza con corte = f(inicial).

Las reexpansiones de IDA\* pueden suponer un elevado coste temporal, existen algoritmos que por lo general reexpanden menos nodos. Su funcionamiento se basa en eliminar los nodos menos prometedores y guardar información que permita reexpandirlos. Ejemplos: Best First recursivo, Memory Bound A\* (MA\*).

*Best First Recursivo*

Es una implementación del Best First recursiva con coste lineal en espacio O(rp). Olvida una rama cuando su coste supera la mejor alternativa, el coste de esta rama olvidada se almacena en el padre como su nuevo coste. La rama es reexpandida si su coste vuelve a ser el mejor (regeneramos toda la rama olvidada).

Por lo general, reexpande menos nodos que IDA\*. El límite de memoria (lineal en espacio) provoca muchas reexpansiones en ciertos problemas. Al no poder controlar los repetidos, su coste en tiempo puede elevarse si hay ciclos. Solución: relajar la restricción de memoria.

*Memory Bound A\* (MA\*)*

Impone un límite de memoria (número de nodos que se pueden almacenar, mayor que O(rp)). Exploramos usando A\* y almacenamos nodos mientras quepan en la memoria. Cuando no quepan eliminamos los peores guardando el mejor coste de los descendientes olvidados. Reexpandimos si los nodos olvidados son mejores. El algoritmo es completo si el camino solución cabe en memoria.

Diferencia entre espacio de estados y espacio de soluciones:

Diagrama

Descripción generada automáticamente

**Búsqueda local:**

Los algoritmos vistos anteriormente solamente sirven cuando se pueda plantear el problema como la búsqueda de un camino en un espacio de estados. Pero puede haber problemas como:

* Este planteamiento es demasiado artificial, ya que no se tiene realmente operadores de cambio de estado, o no hay una noción de coste asociada a los operadores del problema.
* La posibilidad de hallar la solución óptima está fuera de toda posibilidad, dado que el tamaño de búsqueda es demasiado grande y nos conformamos con una solución que podamos considerar buena.

En este tipo de problemas puede ser relativamente fácil hallar una solución inicial, aunque no sea demasiado buena. Esto hace que nos podamos plantear el problema como una búsqueda dentro del espacio de soluciones, en lugar de en el de caminos.

También es posible que sea muy difícil crear una solución inicial (aunque sea mala) y podamos iniciar la búsqueda desde el espacio de no soluciones (soluciones no válidas) suponiendo que la búsqueda nos llevará al final al espacio de soluciones.

(\*) El comenzar desde el espacio de soluciones o desde el de no soluciones dependerá de las características del problema. A veces el número de soluciones válidas es muy reducido debido a que lo que exigimos para ser una solución es muy estricto y es imposible generar una solución inicial sin realizar una búsqueda. Si sabemos que partiendo desde una solución inválida podremos alcanzar el espacio de soluciones, no habrá problema, el algoritmo hará esa búsqueda inicial guiada por la función heurística. Si no es así, deberemos usar otros algoritmos más exhaustivos que nos aseguren hallar una solución.

Queremos la mejor de entre las soluciones posibles alcanzable en un tiempo razonable (el óptimo es imposible). Tendremos una función que nos evalúa la calidad de la solución, pero que no está ligada a ningún coste necesariamente. La búsqueda se realiza desde una solución inicial que intentamos mejorar modificándola (operadores), estos mismos operadores son los que nos mueven entre soluciones vecinas.

La función heurística:

* Aproxima la calidad de una solución (no representa un coste).
* Se ha de optimizar (maximizarla o minimizarla).
* Combinará los elementos del problema y sus restricciones (posiblemente con diferentes pesos).
* No hay ninguna restricción sobre como ha de ser la función, solo ha de representar las relaciones de calidad entre las soluciones.
* Puede tomar valores positivos o negativos.

Los elementos que conforman la función heurística han de tener las mismas unidades (análisis dimensional correcta).

El tamaño del espacio de soluciones por lo general no permite obtener el óptimo, los algoritmos no pueden hacer una exploración sistemática, la función heurística se usará para podar el espacio de búsqueda (soluciones que no merece la pena explorar), no se suele guardar historia del camino recorrido (el gasto de memoria es mínimo) y la falta total de memoria puede suponer un problema (bucles).

El saber cuando hemos acabado la búsqueda dependerá totalmente del algoritmo, que tendrá que decidir cuando ya no es posible encontrar una solución mejor. Por lo tanto, no se tiene un estado final definido.

*Hill Climbing*

Escalada simple -> se busca cualquier operación que suponga una mejora respecto al padre.

Escalada por máxima pendiente -> se selecciona el mejor movimiento (no el 1º de ellos) que suponga mejora respecto al estado actual.

En este algoritmo sólo se consideran los descendientes cuya función de estimación es mejor que la del padre (poda del espacio de búsqueda). Se puede usar una pila y guardar los hijos mejores que el padre para hacer backtracking, pero por lo general es prohibitivo y es posible que el algoritmo no encuentre una solución, aunque la haya.

Las características de la función heurística y la solución inicial determinan el éxito y rapidez de la búsqueda. La estrategia del algoritmo hace que la búsqueda pueda acabar en un punto donde la solución sólo sea la óptima aparentemente.

Problemas:

* Máximo local: ningún vecino tiene mejor coste.
* Meseta: todos los vecinos son iguales.
* Cresta: la pendiente de la función sube y baja (efecto escalón).

Posibles soluciones:

* Reiniciar la búsqueda en otro punto buscando mejorar la solución actual (Random Restarting Hill Climbing).
* Hacer backtracking a un nodo anterior y seguir el proceso en otra dirección.
* Aplicar dos o más operaciones antes de decidir el camino.
* Hacer HC en paralelo (por ejemplo, dividir el espacio de búsqueda en regiones y explorar las más prometedoras, posiblemente compartiendo información).

*Simulated Annealing*

Es un algoritmo de Hill Climbing estocástico (elegimos un sucesor de entre todos los posibles según una distribución de probabilidad, el sucesor podría ser peor). Hacemos paseos aleatorios por el espacio de soluciones. Es inspirado en el proceso físico de enfriamiento controlado, se calienta un metal/disolución a alta temperatura y se enfría progresivamente de manera controlada. Si el enfriamiento es adecuado se obtiene la estructura de menor energía (mínimo global).

Una ventaja de este algoritmo es que no se tiene que generar todos los sucesores de un nodo, basta elegir un sucesor al azar y decidir si continuamos por él o no.

Metodología:

* Identificar los elementos del problema con los del problema físico.
* Temperatura -> parámetro de control del funcionamiento del algoritmo.
* Energía -> calidad de la solución f(n).
* Función de aceptación -> permite decidir si escoger un nodo sucesor.
  + F(∆f, T), función de la temperatura y la diferencia de calidad entre la solución actual y la solución candidata.
  + A menor temperatura menor probabilidad de elegir sucesores peores.

El algoritmo realizará un número total de iteraciones fijo y cada cierto número de ellas el valor de la temperatura disminuirá en cierta cantidad, partiendo de una temperatura inicial y llegando a cero en la última fase. De manera que la elección de estos dos parámetros (número total de iteraciones y número de iteraciones entre cada bajada de temperatura) determinarán el comportamiento del algoritmo.

Gráfico

Descripción generada automáticamente con confianza media

Si el número de pasos es muy pequeño, la temperatura bajará muy rápidamente y el camino explorado será relativamente aleatorio. Si el número de pasos es más grande, la bajada de temperatura será más suave y la búsqueda será mejor.

Adaptable a problemas de optimización combinatoria (configuración óptima de elementos) y continua (punto óptimo en un espacio N-dimensional). Es el algoritmo indicado para problemas grandes en los que el óptimo está rodeado de muchos óptimos locales y para problemas en los que encontrar una heurística discriminante es difícil (una elección aleatoria es tan buena como otra cualquiera).

Problemas: determinar los valores de los parámetros requiere experimentación.

*Algoritmos genéticos*

Inspirado en el mecanismo de selección natural:

* Los seres vivos se adaptan al entorno gracias a las características heredadas de sus progenitores.
* Las posibilidades de supervivencia y reproducción son proporcionales a la bondad de esas características.
* La combinación de buenos individuos puede dar lugar a individuos mejor adaptados.

Podemos trasladar la analogía a la búsqueda local:

* Las soluciones corresponden a individuos.
* La función de calidad indica la bondad de la solución.
* Combinando buenas soluciones podemos obtener soluciones mejores.

Resolver un problema mediante AAGG requiere:

* Dar una codificación a las características de las soluciones (por ejemplo, una cadena binaria), por analogía a esta codificación se le denomina gen.
* Tener una función que mida la calidad de la solución parecida a la función heurística (función de fitness).
* Disponer de operadores que combinen las soluciones para obtener nuevas soluciones (operadores de crossover).
* Decidir el número de individuos inicial (población inicial).
* Decidir una estrategia para hacer la combinación de individuos.

El uso de los algoritmos genéticos presenta ciertas ventajas prácticas respecto a los algoritmos anteriores. Estamos redefiniendo como se representan los problemas, todos tendrán una representación común independientemente de su naturaleza, por lo que solo nos deberemos de preocupar de como codificar el estado, su implementación viene determinada por esa codificación. Tampoco tendremos que preocuparnos de los operadores ya que esa representación común nos determina los posibles operadores que podremos usar y serán comunes para todos los problemas.

Codificación:

Lo más habitual es transformarlos a una cadena de bits. Cada bit o grupo de bits codificará una característica de la solución.

La ventaja de esta codificación es que los operadores de modificación de la solución son muy sencillos de implementar. También nos da una aproximación del tamaño del espacio de búsqueda, ya que el número de soluciones posibles corresponderá al mayor número binario que podamos representar con la codificación.

Operadores:

La combinación de individuos se realiza mediante operadores de cruce, el operador básico es el cruce por un punto (crossover):

* Se elige aleatoriamente un punto de la codificación.
* La información de dos individuos se combina usando ese punto como referencia.

Existen otras posibilidades:

* Cruce en 2 puntos.
* Intercambio aleatorio de bits.
* Operadores adhoc según la representación.

Operadores de mutación:

* Por analogía con la combinación de genes, a veces la información de parte de ellos cambia aleatoriamente.
* El operador básico de mutación consiste en cambiar el signo de un bit con cierta probabilidad.

Por lo general la probabilidad de mutación es mucho más pequeña que la probabilidad de cruce. La elección de estos valores es crítica para el correcto funcionamiento del algoritmo y muchas veces depende del problema.

Si la probabilidad de cruce es muy alta, cada generación tendrá muchos individuos nuevos, esto puede hacer que la exploración vaya más deprisa, pero también puede hacer que los patrones que llevan a soluciones mejores no puedan estabilizarse.

Si la probabilidad es muy pequeña, la exploración será más lenta y puede tardar mucho en converger.

Con la mutación pasa algo similar, esta es esencial para que el algoritmo pueda explorar en partes del espacio de búsqueda que no son alcanzables con la combinación de las soluciones iniciales, pero una probabilidad muy alta puede hacer que la exploración no converja al introducir demasiado ruido en la población.

Combinación de individuos

Cada paso de búsqueda es una generación de individuos, su tamaño se mantiene constante (N). Para pasar a la siguiente generación debemos elegir que individuos se han de combinar (generación intermedia).

Elección de los individuos:

* Cada individuo se elige con probabilidad proporcional a su calidad.
* Se establecen N torneos aleatorios entre parejas de individuos, se eligen los que ganan en cada torneo.
* Se define un ranking lineal entre individuos según su función de calidad.

Siempre habrá individuos que aparezcan más de una vez e individuos que no aparezcan.

El primer criterio de selección corre el peligro de degenerar en pocas generaciones a una población homogénea, acabando en un óptimo local.

Los otros dos son algo más complejos, pero aseguran cierta diversidad de individuos en la población. (al asegurar la variedad en la población permite encontrar mejores soluciones)

Población -> conjunto de soluciones que tenemos en un momento específico.

Generación -> cada iteración de un algoritmo genético.

Generación intermedia -> forma habitual de pasar de una generación a otra y compuesta por las parejas que se van a combinar.

Algoritmo genético canónico

Los pasos que realiza el AG básico son estos:

1. Se escogen N individuos de la generación actual para la generación intermedia. (Según el criterio escogido)
2. Se emparejan los individuos y para cada pareja:
   1. Con una probabilidad (P\_cruce) se aplica el operador de cruce a los individuos y se obtienen dos nuevos individuos. Y con una probabilidad 1 – P\_cruce se mantienen la pareja original en la siguiente generación.
   2. Con una probabilidad (P\_mutación) se mutan los nuevos individuos.
3. Estos individuos forman la nueva generación.
4. Iterar hasta que la población converja (la función de calidad no mejora) o pase un número específico de iteraciones.

La probabilidad de cruce influirá en la variedad de la nueva generación, cuanto más baja sea, más parecida será a la generación anterior y harán falta más iteraciones para converger. Y la probabilidad de mutación siempre es muy pequeña para evitar una búsqueda aleatoria.

Aplicación

Aplicable casi a cualquier tipo de problema. Permite abordar problemas para los que no se dispone de una función heurística adecuada y por lo general serán peores que un algoritmo clásico con una buena heurística.

Problemas: Codificación de los estados, determinar los parámetros del algoritmo (tamaño de la población, iteraciones, probabilidad de cruce y mutación). Y en algún tipo de problemas pueden no funcionar muy bien.

**Búsqueda con adversario (juegos):**

Uso: decidir mejor jugada en cada momento para cierto tipo de juegos.

Hay diferentes tipos de juegos según sus características:

* Número de jugadores, toda la información conocida por todos los jugadores, azar, indeterminismo, cooperación/competición, recursos limitados…

Nos focalizaremos en juegos con:

* 2 jugadores (2 agentes).
* Movimientos alternos (jugador MAX, jugador MIN).
* Información perfecta (no hay ocultación de información y no interviene el azar).
* Ejemplo: ajedrez, damas, …

Representación del juego

Puede ser definido como un problema de espacio de estados:

* Estado = elementos del juego (indica las características del juego en un instante específico).
* Estado inicial = determina desde donde se empieza a jugar y quién inicia el juego.
* Estados finales = estados ganadores (definidos por sus propiedades / características)
* Acciones / operadores = reglas del juego.

Son problemas con características especiales:

* La accesibilidad de los estados depende de las acciones elegidas por el contrario.
* Dos tipos de soluciones diferentes (una para cada jugador).
* No hay noción de optimalidad (todas las soluciones son iguales, no importa la longitud del camino).

Una aproximación trivial

La aproximación trivial es generar todo el árbol de jugadas (explorar todas las posibles jugadas que puede hacer cada jugador hasta encontrar un camino que lleve a un estado final) y< etiquetar las jugadas terminales dependiendo de si gana MAX o MIN (+1 o -1). El objetivo es encontrar un conjunto de movimientos accesible que dé como ganador a MAX. Se propagan los valores de las jugadas terminales de las hojas hasta la raíz, elegimos una roma de una hoja ganadora accesible. Una búsqueda en profundidad minimiza el espacio. Pero en juegos mínimamente complejos esta búsqueda es impracticable. (ajedrez -> O(2^35))

La exploración completa será exponencial respecto al número de posibles jugadas en cada nivel (r) y la profundidad de la solución (p) -> O(r^p)

Una aproximación heurística

Definir una función que nos indique lo cerca que estamos de una jugada ganadora (o perdedora). En esta función intervendrá información del dominio. No representa ningún coste ni es una distancia en pasos. Por convención las jugadas ganadoras se evalúan a +∞ y las perdedoras a -∞. El algoritmo busca con profundidad limitada y sólo decide la siguiente jugada a partir del nodo raíz, cada nueva decisión implicará repetir la búsqueda. A mayor profundidad en la búsqueda mejor jugaremos.

Algoritmo MinMax

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Una aproximación aún mejor

MinMax con poda αβ

Diagrama

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación, Word

Descripción generada automáticamente

Imagen que contiene Interfaz de usuario gráfica

Descripción generada automáticamente

El ratio de reexpansiones a nodos nuevos para la profundidad iterativa tiene a cero con el aumento del factor de ramificación, por lo que el precio que hay que pagar en nodos extras explorados es reducido, si tenemos más posibilidades de llegar a alcanzar la complejidad media de O(r^p/2).

**Satisfacción de restricciones:**

Componentes del estado:

* Variables (el problema se puede representar mediante un conjunto de variables).
* Dominios (valores posibles para cada variable), puede ser un conjunto finito de valores discretos o intervalos de valores continuos.
* Restricciones de consistencia binarias (o n-arias) entre las variables.

Objetivo:

* Encontrar un estado que satisface las restricciones (asignación de valores a las variables, que satisfaga las restricciones).

Representación:

Estado = grafo de restricciones:

* Variables = etiquetas de nodos.
* Dominios = contenido de nodos.
* Restricciones = arcos dirigidos y etiquetados entre nodos.

Algoritmos:

* Generación y prueba: muy ineficiente (con n variables y m posibles valores, el espacio de búsqueda sería O(m^n)).

No hay noción de calidad de solución, ya que todas las asignaciones son igual de buenas y lo único que las diferencian es si satisfacen las restricciones o no, por lo que no podemos confiar en una función heurística que nos indique que combinaciones debemos mirar primero, o como modificar una asignación para acercarla a la solución. Esto nos deja con que los únicos algoritmos que podremos utilizar son los de búsqueda no informada, lo que hará que tengamos que estudiar con más profundidad el problema para poder solucionarlo de una manera eficiente.

* Búsqueda ciega: búsqueda en profundidad con backtracking cronológico.
* Propagación de restricciones:
  + Antes de la búsqueda.
  + Durante la búsqueda.

*Backtracking Cronológico*

Búsqueda en profundidad sobre las variables (búsqueda no informada).

Asignar valor por estrategia exhaustiva:

* Comprobar restricciones tras cada posible asignación.
* Si no se satisfacen para ningún valor, es decir, en el momento que una solución parcial viole alguna de las restricciones del problema se hace backtracking sobre la última asignación válida (volver hacia atrás).

La búsqueda se realiza en el espacio de soluciones parciales.

Tipos de variables:

* Pasadas: las que ya tienen un valor asignado y forman parte de la solución parcial.
* Actual: la que debemos asignarle un valor.
* Futuras: las que aún no tienen valor.

*Propagación de restricciones*

Un conjunto de restricciones puede incluir otras que estaban implícitas. La propagación de restricciones es el proceso de hacerlas explícitas.

El papel de la PR es disminuir el espacio de búsqueda, descartando aquellas combinaciones de valores que no pueden aparecer en ninguna solución. También sirven para reducir el número de vueltas atrás inútiles que realiza el algoritmo de backtracking cronológico. Debemos realizar la propagación:

1. Preproceso (eliminar zonas del espacio donde no hay soluciones).
2. Durante el proceso: podar el espacio a medida que la búsqueda progresa (Forward Checking).

Cada ciclo tiene dos partes:

1. Se propagan las restricciones: Se podrían utilizar de reglas de inferencia. Tener en cuenta que las restricciones no tienen porqué ser independientes (muchas restricciones implican a varias variables, una variable participa en muchas restricciones).
2. Se analiza el resultado: solución encontrada, solución imposible o seguir buscando (proceso heurístico de búsqueda).

Estas técnicas se han desarrollado a partir de heurísticas derivadas de propiedades que tienen los grafos de restricciones, que son:

* K-consistencia: se dice que un grafo de restricciones es k-consistente si para cada variable Xi del grafo, cada conjunto de k variables que la incluya y cada valor del dominio de Xi podemos encontrar una combinación de k valores de estas variables que no viola las restricciones entre ellas. La ventaja de estas propiedades es que se pueden comprobar con algoritmos de coste polinómico. Poda de valores que no sean posibles para un grupo de k variables.
* K-consistencia con k = 2 -> arco-consistencia: se dice que un grafo de restricciones es arco-consistente, si para cada pareja de variables del grafo, para cada uno de los valores de una de las variables, podemos encontrar otro valor de la otra variable que no viola las restricciones entre ellas. Esta propiedad permite eliminar valores del dominio de una variable. Si dado un valor de una variable y dada otra variable con la que tenga una restricción, no podemos encontrar algún valor en la otra variable que no viole esa restricción, entonces podemos eliminar el valor, ya que no puede formar parte de una solución.

El objetivo será pues, a partir de un grafo de restricciones, eliminar todos aquellos valores que no hagan arco consistente el grafo, de manera que reduzcamos el número de posibles valores que pueden tener las variables del problema y por lo tanto el número de combinaciones a explorar.

La arco-consistencia no siempre nos asegura una reducción en los dominios del problema.

La k-consistencia para el caso de k = 3, se denomina camino consistencia, y permite eliminar más combinaciones, pero con un coste mayor.

*Forward Checking*

Modificación del algoritmo de búsqueda en profundidad con backtracking cronológico (introducimos la propagación de restricciones después de cada asignación, eliminando de esta manera valores de las variables que quedan por asignar). Es decir, es la combinación de la búsqueda con backtracking y la propagación de restricciones.

Anticipación: detectar cuanto antes caminos sin solución y podarlos.

* Asignar un valor y consultar las restricciones sobre las variables futuras con arco desde la actual.
* Se eliminan valores no compatibles de los dominios correspondientes a dichas variables futuras.

La ventaja es que podremos hacer backtracking antes de que lleguemos a la variable que acabará provocándolo (anticipación).

Equivale a hacer arco-consistente la variable actual con las futuras en cada paso.

La eficiencia dependerá del problema (incrementamos el coste de cada iteración).

Otras heurísticas

La búsqueda con backtracking puede mejorarse:

* Comprobando consistencias más restrictivas (con mayor coste).
  + Haciendo arco consistente todo el problema a cada paso (algoritmo MAC).
* Escogiendo el orden de prueba de las variables.
  + ¿Cuándo?
    - Antes de la búsqueda (orden fijo).
    - Durante la búsqueda (orden variable).
  + ¿Qué orden?
    - Primero variables con más restricciones.
    - Primero variables con menos valores.
  + La reordenación de variables puede reducir el tiempo de búsqueda varios órdenes de magnitud en ciertos problemas.