

APUNTS DE RESOLUCIÓ DE PROBLEMES

Departament de Llenguatges i Sistemes Informàtics



FIB

Facultat d'Informàtica
de Barcelona

UNIVERSITAT POLITÈCNICA DE CATALUNYA

This work is licensed under the Creative Commons
Attribution-NonCommercial-ShareAlike License. 

To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/2.0/> or
send a letter to:

Creative Commons,
559 Nathan Abbott Way, Stanford,
California 94305,
USA.

0. Introducción	1
I Resolución de problemas	3
1. Resolución de problemas	5
1.1. ¿Qué es un problema?	5
1.2. El espacio de estados	6
1.3. Algoritmos de búsqueda en el espacio de estados	9
2. Búsqueda no informada	13
2.1. Búsqueda independiente del problema	13
2.2. Búsqueda en anchura prioritaria	13
2.3. Búsqueda en profundidad prioritaria	14
2.4. Búsqueda en profundidad iterativa	16
2.5. Ejemplos	18
3. Búsqueda heurística	23
3.1. El conocimiento importa	23
3.2. El óptimo está en el camino	23
3.3. Tú primero y tú después	24
3.4. El algoritmo A*	24
3.5. Pero, ¿encontraré el óptimo?	27
3.5.1. Admisibilidad	27
3.5.2. Consistencia	28
3.5.3. Heurístico más informado	29
3.6. Mi memoria se acaba	29
3.6.1. El algoritmo IDA*	30
3.6.2. Otras alternativas	31
4. Búsqueda local	35
4.1. El tamaño importa, a veces	35
4.2. Tu sí, vosotros no	36
4.3. Un mundo de posibilidades	38
4.4. Demasiado calor, demasiado frío	40
4.5. Cada oveja con su pareja	42
4.5.1. Codificación	43
4.5.2. Operadores	43
4.5.3. Combinación de individuos	44
4.5.4. El algoritmo genético canónico	45
4.5.5. Cuando usarlos	46

5. Búsqueda con adversario	47
5.1. Tú contra mi o yo contra ti	47
5.2. Una aproximación trivial	48
5.3. Seamos un poco más inteligentes	48
5.4. Seamos aún más inteligentes	50
6. Satisfacción de restricciones	55
6.1. De variables y valores	55
6.2. Buscando de manera diferente	56
6.2.1. Búsqueda con backtracking	57
6.2.2. Propagación de restricciones	59
6.2.3. Combinando búsqueda y propagación	61
6.3. Otra vuelta de tuerca	62
6.4. Ejemplo: El Sudoku	63

0. Introducción

Este documento contiene las notas de clase del tema de resolución de problemas y algoritmos de búsqueda de la asignatura Intel·ligència Artificial del grado en informàtica de la Facultat d'Informàtica de Barcelona.

El documento está pensado como complemento a las transparencias utilizadas en las clases. El objetivo es que se lea esta documentación antes de la clase correspondiente para que se obtenga un mejor aprovechamiento de las clases y se puedan realizar preguntas y dinamizar la clase.

En ningún caso se pueden tomar estos apuntes como un sustituto de las clases de teoría.

Parte I

Resolución de problemas

1.1 ¿Qué es un problema?

Una de las principales capacidades de la inteligencia humana es su capacidad para resolver problemas. La habilidad para analizar los elementos esenciales de cada problema, abstrayéndolos, el identificar las acciones que son necesarias para resolverlos y el determinar cual es la estrategia más acertada para atacarlos, son rasgos fundamentales que debe tener cualquier entidad inteligente. Es por eso por lo que la resolución de problemas es uno de los temas básicos en inteligencia artificial.

Podemos definir la resolución de problemas como el proceso que partiendo de unos datos iniciales y utilizando un conjunto de procedimientos escogidos a priori, es capaz de determinar el conjunto de pasos o elementos que nos llevan a lo que denominaremos una solución. Esta solución puede ser, por ejemplo, el conjunto de acciones que nos llevan a cumplir cierta propiedad o como deben combinarse los elementos que forman los datos iniciales para cumplir ciertas restricciones.

Evidentemente, existen muchos tipos diferentes de problemas, pero todos ellos tienen elementos comunes que nos permiten clasificarlos y estructurarlos. Por lo tanto, no es descabellada la idea de poder resolverlos de manera automática, si somos capaces de expresarlos de la manera adecuada.

Para que podamos resolver problemas de una manera automatizada es necesario, en primer lugar, que podamos expresar las características de los problemas de una manera formal y estructurada. Eso nos obligará a encontrar un lenguaje común que nos permita definir problemas.

En segundo lugar, hemos de definir algoritmos que representen estrategias que nos permitan hallar la solución de esos problemas, que trabajen a partir de ese lenguaje de representación y que nos garanticen hasta cierta medida que la solución que buscamos será hallada.

Si abstraemos las características que describen un problema podemos identificar los siguientes elementos:

1. Un punto de partida, los elementos que definen las características del problema.
2. Un objetivo a alcanzar, qué queremos obtener con la resolución.
3. Acciones a nuestra disposición para resolver el problema, de qué herramientas disponemos para manipular los elementos del problema.
4. Restricciones sobre el objetivo, qué características debe tener la solución
5. Elementos que son relevantes en el problema definidos por el dominio concreto, qué conocimiento tenemos que nos puede ayudar a resolver el problema de una manera eficiente.

Escogiendo estos elementos como base de la representación de un problema podemos llegar a diferentes aproximaciones, algunas generales, otras más específicas. De entre los tipos de aproximaciones podemos citar:

Espacio de estados: Se trata de la aproximación más general, un problema se divide en un conjunto de pasos de resolución que enlazan los elementos iniciales con los elementos que describen la solución, donde en cada paso podemos ver como se transforma el problema.

Reducción a subproblemas: En esta aproximación suponemos que podemos descomponer el problema global en problemas más pequeños de manera recursiva hasta llegar a problemas simples. Es necesario que exista la posibilidad de realizar esta descomposición.

Satisfacción de restricciones: Esta aproximación es específica para problemas que se puedan plantear como un conjunto de variables a las que se han de asignar valores cumpliendo ciertas restricciones.

Juegos: También es una aproximación específica, en la que el problema se plantea como la competición entre dos o más agentes

La elección de la forma de representación dependerá de las características del problema, en ocasiones una metodología especializada puede ser más eficiente.

1.2 El espacio de estados

La aproximación más general y más sencilla de plantear es la que hemos denominado *espacio de estados*. Debemos suponer que podemos definir un problema a partir de los elementos que intervienen en él y sus relaciones. En cada instante de la resolución de un problema estos elementos tendrán unas características y relaciones específicas que son suficientes para determinar el momento de la resolución en el que estamos. Denominaremos **estado** a la representación de los elementos que describen el problema en un momento dado. Distinguiremos dos estados especiales, el **estado inicial** (punto de partida) y el **estado final** (objetivo del problema).

La cuestión principal que nos deberemos plantear será qué debemos incluir en el estado. No existen unas directrices que nos resuelvan esta cuestión, pero podemos tomar como línea general que debemos incluir lo suficiente para que, cuando obtengamos la solución, ésta pueda trasladarse al problema real, ya que una simplificación excesiva nos llevará a soluciones inservibles. En el otro extremo, representar elementos de estado que sean irrelevantes o superfluos lo único que conseguirá es aumentar innecesariamente el coste computacional de la resolución.

Para poder movernos entre los diferentes estados que definen el problema, necesitaremos lo que denominaremos **operadores de transformación**. Un operador es una función de transformación sobre la representación de un estado que lo convierte en otro estado. Los operadores definirán una **relación de accesibilidad** entre estados.

Los elementos que definen un operador son:

1. **Condiciones de aplicabilidad**, las condiciones que debe cumplir un estado para que podamos aplicárselo.
2. **Función de transformación**, la transformación que ha de aplicarse al estado para conseguir un estado sucesor al actual.

Igual que con la elección de los elementos que forman parte del estado, tampoco existen unas directrices que nos permitan decidir, de manera general, que operadores serán necesarios para resolver el problema, cuantos serán necesarios o que nivel de granularidad han de tener (como de diferente es el estado transformado respecto al original). Se puede dar el mismo consejo que con los estados, pocos operadores pueden hacer que nuestro problema sea irresoluble o que la solución no nos sirva en el problema real, demasiados operadores pueden hacer que el coste de la resolución sea prohibitivo.

Los estados y su relación de accesibilidad conforman lo que se denomina el **espacio de estados**. Éste representa todos los caminos que hay entre todos los estados posibles de un problema. Podría asimilarse con un mapa de carreteras de un problema, la solución de nuestro problema está dentro de ese mapa, solo nos falta hallar el camino adecuado.

El último elemento que nos queda por definir es la **solución**. La definiremos de dos maneras, como la secuencia de pasos que llevan del estado inicial al final (secuencia de operadores) o el estado final del problema. Existen problemas en los que la secuencia de operadores es el objetivo de la solución,

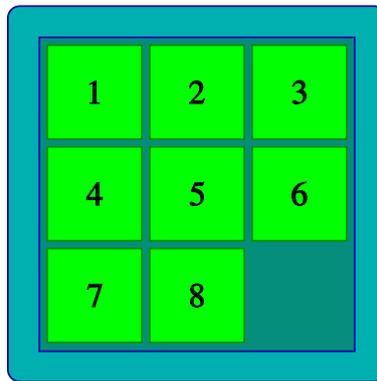
en éstos, por lo general, sabemos como es la solución, pero no sabemos como construirla y existen también problemas en los que queremos saber si es posible combinar los elementos del problema cumpliendo ciertas restricciones, pero la forma de averiguarlo no es importante.

También podremos catalogar los diferentes tipos problema según el tipo de solución que busquemos, dependiendo de si nos basta con una cualquiera, queremos la mejor o buscamos todas las soluciones posibles. Evidentemente el coste computacional de cada uno de estos tipos es muy diferente.

En el caso de plantearnos el cuantos recursos gastamos para obtener una solución, deberemos introducir otros elementos en la representación del problema. Definiremos el **coste de una solución** como la suma de los costos individuales de la aplicación de los operadores a los estados, esto quiere decir que cada operador tendrá también asociado un coste.

Vamos a ver a continuación dos ejemplos de como definir problemas como espacio de estados:

Ejemplo 1.1 *El ocho puzzle es un problema clásico que consiste en un tablero de 9 posiciones dispuesto como una matriz de 3×3 en el que hay 8 posiciones ocupadas por fichas numeradas del 1 al 8 y una posición vacía. Las fichas se pueden mover ocupando la posición vacía, si la tienen adyacente. El objetivo es partir de una disposición cualquiera de las fichas, para obtener una disposición de éstas en un orden específico. Tenemos una representación del problema en la siguiente figura:*



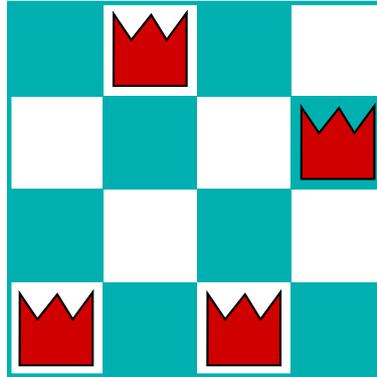
La definición de los elementos del problema para plantearlo en el espacio de estados podría ser la siguiente:

- *Espacio de estados: Configuraciones de 8 fichas en el tablero*
- *Estado inicial: Cualquier configuración*
- *Estado final: Fichas en un orden específico*
- *Operadores: Mover el hueco*
 - *Condiciones: El movimiento está dentro del tablero*
 - *Transformación: Intercambio entre el hueco y la ficha en la posición del movimiento*
- *Solución: Que movimientos hay que hacer para llegar al estado final, posiblemente nos interesa la solución con el menor número de pasos*

En esta definición hemos escogido como operador mover el hueco, evidentemente, en el problema real lo que se mueven son las fichas, pero elegir el movimiento de una ficha como operador nos habría dado 8 posibles aplicaciones con 4 direcciones posibles para cada una. Al elegir el mover hueco como operador tenemos una única aplicación con 4 direcciones posibles.

Este tipo de elecciones sobre como representar el problema pueden hacer que sea computacionalmente más o menos costoso de resolver, ya que estamos variando la forma y el tamaño del espacio de estados.

Ejemplo 1.2 *El problema de las N reinas es también un problema clásico, en este caso el problema se trata de colocar N reinas en un tablero de ajedrez de $N \times N$ de manera que no se maten entre sí. En este problema no nos interesa la manera de hallar la solución, sino el estado final. En la figura tenemos representada la solución para un problema con dimensión 4:*



La definición de los elementos del problema para plantearlo en el espacio de estados podría ser la siguiente:

- *Espacio de estados: Configuraciones de 0 a n reinas en el tablero con solo una por fila y columna*
- *Estado inicial: Configuración sin reinas en el tablero*
- *Estado final: Configuración en la que ninguna reina se mata entre sí*
- *Operadores: Colocar una reina en una fila y columna*
 - *Condiciones: La reina no es matada por ninguna otra ya colocada*
 - *Transformación: Colocar una reina más en el tablero en una fila y columna determinada*
- *Solución: Una solución, pero no nos importan los pasos*

También podríamos haber hecho otras elecciones de representación como por ejemplo que el estado inicial tuviera las N reinas colocadas, o que el operador permitiera mover las reinas a una celda adyacente. Todas estas alternativas supondrían un coste de solución más elevado.

Un elemento que será importante a la hora de analizar los problemas que vamos a solucionar, y que nos permitirá tomar decisiones, es el *tamaño del espacio de búsqueda* y su *conectividad*. Este tamaño influirá sobre el tipo de solución que podemos esperar, por ejemplo, si el tamaño es demasiado grande, encontrar la solución óptima puede ser irrealizable, el tipo de algoritmo que es más adecuado, por ejemplo, habrá algoritmos más aptos para soluciones que están más lejos del estado inicial, y cuál será el coste computacional que implicará la resolución del problema.

Es esencial entonces, a la hora de plantear un problema, estimar cual es el **número de estados** que contiene. Por ejemplo, en el caso del ocho puzzle tenemos tantas combinaciones como posibles ordenes podemos hacer de las 8 piezas mas el hueco, esto significa $9!$ estados posibles (362880 estados). Probablemente no tendremos que recorrer todos los estados posibles del problema, pero nos puede dar una idea de si el problema será mas o menos difícil de resolver¹.

Otro elemento a considerar es la **conectividad entre los estados**, que se puede calcular a partir del factor de ramificación de los operadores que podemos utilizar. No es lo mismo recorrer un espacio

¹Se solía comentar hace unos años que problemas que tengan menos de 2^{32} estados para los que solo nos interesa saber si existe una estado solución eran la frontera de los problemas sencillos y que se pueden resolver por fuerza bruta simplemente enumerándolos todos, con la capacidad de cálculo actual probablemente ahora ese número mínimo de estados sea algo mayor.

Algoritmo 1.1 Esquema de algoritmo de búsqueda en espacio de estados

Función: Búsqueda en espacio de estados()
Datos: El estado inicial
Resultado: Una solution
 Seleccionar el primer estado como el estado actual
mientras *estado actua* \neq *estado final* **hacer**
 | Generar y guardar sucesores del estado actual (expansión)
 | Escoger el siguiente estado entre los pendientes (selección)
fin

de estados en el que de un estado podemos movernos a unos pocos estados sucesores, que un espacio de estados en los que cada estado esta conectado con casi todos los estados posibles.

Esta es la razón por la que elegir operadores con un factor de ramificación no muy grande es importante. Pero debemos tener en cuenta otra cosa, un factor de ramificación pequeño puede hacer que el camino hasta la solución sea más largo. Como siempre, hemos de buscar el compromiso, tener mucha conectividad es malo, pero muy poca también puede ser malo.

En el caso del ocho puzzle, el factor de ramificación es 4 en el peor de los casos, pero en media podemos considerar que estará alrededor de 2.



Puedes tomarte unos minutos en pensar otros problemas que puedas definir según el paradigma del espacio de estados. Deberás fijarte en que los problemas representables de este modo se pueden reducir a la búsqueda de un camino.

1.3 Algoritmos de búsqueda en el espacio de estados

La resolución de un problema planteado como un espacio de estados pasa por la exploración de éste. Debemos partir del estado inicial marcado por el problema, evaluando cada paso posible y avanzando hasta encontrar un estado final. En el caso peor deberemos explorar todos los posibles caminos desde el estado inicial hasta poder llegar a un estado que cumpla las condiciones del estado final.

Para poder implementar un algoritmo capaz de explorar el espacio de estados en busca de una solución primero debemos definir una representación adecuada de éste. Las estructuras de datos más adecuadas para representar el espacio de estados son los grafos y los árboles. En estas estructuras cada nodo de representará un estado del problema y los operadores de cambio de estado estarán representados por los arcos. La elección de un árbol o un grafo la determinará la posibilidad de que a un estado solo se pueda llegar a través de un camino (árbol) o que haya diferentes caminos que puedan llevar al mismo estado (grafo).

Los grafos sobre los que trabajarán los algoritmos de resolución de problemas son diferentes de los algoritmos habituales sobre grafos. La característica principal es que el grafo se construirá a medida que se haga la exploración, dado que el tamaño que puede tener hace imposible que se pueda almacenar en memoria (el grafo puede ser incluso infinito). Esto hace que, por ejemplo, los algoritmos habituales de caminos mínimos en grafos no sirvan en este tipo de problemas.

Nosotros nos vamos a centrar en los algoritmos que encuentran una solución, pero evidentemente extenderlo a todas las soluciones es bastante sencillo. En el algoritmo 1.1 se puede ver el que sería un esquema a alto nivel de un algoritmo que halla una solución de un problema en el espacio de estados.

En este algoritmo tenemos varias decisiones que nos permitirán obtener diferentes variantes que

Algoritmo 1.2 Esquema general de búsqueda**Algoritmo:** Busqueda General

Est_abiertos.insertar(Estado inicial)

Actual ← Est_abiertos.primeros()

mientras no *es_final?(Actual)* **y** no *Est_abiertos.vacia?()* **hacer**

Est_abiertos.borrar_primeros()

Est_cerrados.insertar(Actual)

Hijos ← generar_sucesores(Actual)

Hijos ← tratar_repetidos(Hijos, Est_cerrados, Est_abiertos)

Est_abiertos.insertar(Hijos)

Actual ← Est_abiertos.primeros()

fin

se comportarán de distinta manera y que tendrán propiedades específicas. La primera decisión es el **orden de expansión**, o lo que es lo mismo, el orden en el que generamos los sucesores de un nodo. La segunda decisión es el **orden de selección**, o sea, el orden en el que decidimos explorar los nodos que aún no hemos visitado.

Entrando un poco más en los detalles de funcionamiento del algoritmo, podremos distinguir dos tipos de nodos, los **nodos abiertos**, que representarán a los estados generados pero aún no visitados y los **nodos cerrados**, que corresponderán a los estados visitados y que ya se han expandido.

Nuestro algoritmo siempre tendrá una estructura que almacenará los nodos abiertos. Las diferentes políticas de inserción de esta estructura serán las que tengan una influencia determinante sobre el tipo de búsqueda que hará el algoritmo.

Dado que estaremos explorando grafos, y que por lo tanto durante la exploración nos encontraremos con estados ya visitados, puede ser necesario tener una estructura para almacenar los nodos cerrados. Merecerá la pena si el número de nodos diferentes es pequeño respecto al número de caminos, pero puede ser prohibitivo para espacios de búsqueda muy grandes.

En el algoritmo 1.2 tenemos un algoritmo algo más detallado que nos va a servir como esquema general para la mayoría de los algoritmos que iremos viendo.

Variando la estructura de abiertos variamos el comportamiento del algoritmo (orden de visita de los nodos). La función **generar_sucesores** seguirá el orden de generación de sucesores definido en el problema. Este puede ser arbitrario o se puede hacer alguna ordenación usando conocimiento específico del problema. El tratamiento de repetidos dependerá de cómo se visiten los nodos, en función de la estrategia de visitas puede ser innecesario.

Para poder clasificar y analizar los algoritmos que vamos a tratar, escogeremos unas propiedades que nos permitirán caracterizarlos. Estas propiedades serán:

- **Completitud:** Si un algoritmo es completo tenemos la garantía de que hallará una solución de existir, si no lo es, es posible que algoritmo no acabe o que sea incapaz de encontrarla.
- **Complejidad temporal:** Coste temporal de la búsqueda en función del tamaño del problema, generalmente una cota que es función del factor de ramificación y la profundidad a la que se encuentra la solución.
- **Complejidad espacial:** Espacio requerido para almacenar los nodos pendientes de explorar, generalmente una cota que es función del factor de ramificación y la profundidad a la que se encuentra la solución. También se puede tener en cuenta el espacio para almacenar los nodos ya explorados si es necesario para el algoritmo

- **Optimalidad:** Si es capaz de encontrar la mejor solución según algún criterio de preferencia o coste.

Atendiendo a estos criterios podemos clasificar los algoritmos principales que estudiaremos en:

- **Algoritmos de búsqueda no informada:** Estos algoritmos no tienen en cuenta el coste de la solución durante la búsqueda. Su funcionamiento es sistemático, siguen un orden de visitas de nodos fijo, establecido por la estructura del espacio de búsqueda. Los principales ejemplos de estos algoritmos son el de anchura prioritaria, el de profundidad prioritaria y el de profundidad iterativa.
- **Algoritmos de búsqueda heurística y búsqueda local:** Estos algoritmos utilizan una estimación del coste o de la calidad de la solución para guiar la búsqueda, de manera que se basan en algún criterio heurístico dependiente del problema. Estos algoritmos no son sistemáticos, de manera que el orden de exploración no lo determina la estructura del espacio de búsqueda sino el criterio heurístico. Algunos de ellos tampoco son exhaustivos y basan su menor complejidad computacional en ignorar parte del espacio de búsqueda.

Existen bastantes algoritmos de este tipo con funcionamientos muy diferentes, no siempre garantizan el encontrar la solución óptima, ni tan siquiera el hallar una solución. Los principales ejemplos de estos algoritmos son A*, IDA*, Branch & Bound, Hill-climbing. Desarrollaremos estos algoritmos en los capítulos siguientes.



Este es un buen momento para repasar los diferentes algoritmos de recorrido de árboles y grafos y de búsqueda de caminos en grafos que se te han explicado en otras asignaturas. A pesar de que traten con estructuras finitas los principios básicos son muy parecidos.

2.1 Búsqueda independiente del problema

Los algoritmos de búsqueda no informada (también conocidos como algoritmos de búsqueda ciega) no dependen de información propia del problema a la hora de resolverlo. Esto quiere decir que son algoritmos generales y por lo tanto se pueden aplicar en cualquier circunstancia.

Estos algoritmos se basan en la estructura del espacio de estados y determinan estrategias sistemáticas para su exploración. Es decir, siguen una estrategia fija a la hora de visitar los nodos que representan los estados del problema. Se trata también de algoritmos exhaustivos, de manera que pueden acabar recorriendo todos los nodos del problema para hallar la solución.

Existen básicamente dos políticas de recorrido de un espacio de búsqueda, en anchura y en profundidad. Todos los algoritmos que se explicarán a continuación se basan en una de las dos.

Al 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. Su ventaja es que no nos hace falta obtener ningún conocimiento adicional sobre el problema, por lo que siempre son aplicables.

2.2 Búsqueda en anchura prioritaria

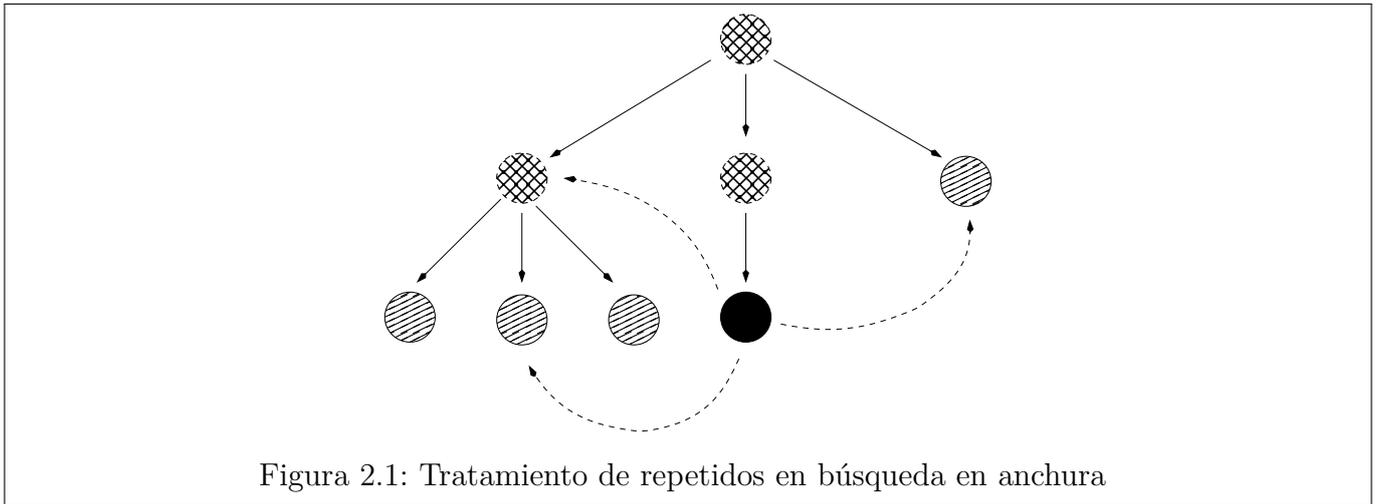
La búsqueda en anchura prioritaria intenta explorar el espacio de búsqueda haciendo un recorrido por niveles, de manera que un nodo se visita solamente cuando todos sus predecesores y sus hermanos anteriores en orden de generación ya se han visitado.

Para obtener este algoritmo solo hemos de instanciar en el algoritmo que vimos en el capítulo anterior la estructura que guarda los nodos abiertos a una cola en el algoritmo general de búsqueda que hemos visto en el capítulo anterior. Esta estructura nos consigue que se visiten los nodos en el orden que hemos establecido.

Si nos fijamos en las propiedades que hemos escogido para clasificar los algoritmos:

- Complejidad: El algoritmo siempre encuentra una solución de haberla.
- Complejidad temporal: Si tomamos como medida del coste el número de nodos explorados, este crece de manera exponencial respecto al factor de ramificación y la profundidad de la solución $O(r^p)$.
- Complejidad espacial: Dado que tenemos que almacenar todos los nodos pendientes por explorar, el coste es exponencial respecto al factor de ramificación y la profundidad de la solución $O(r^p)$.
- Optimalidad: La solución obtenida es óptima respecto al número de pasos desde la raíz, si los operadores de búsqueda tienen coste uniforme, el coste de la solución sería el óptimo.

Respecto al tratamiento de nodos repetidos, si nos fijamos en la figura 2.1 podemos ver los diferentes casos con los que nos encontramos.



El nodo de color negro sería el nodo generado actual, los nodos en cuadrícula son nodos cerrados y los nodos rayados son nodos abiertos. Si el nodo generado actual está repetido en niveles superiores (más cerca de la raíz), su coste será peor ya que su camino desde la raíz es más largo, si está al mismo nivel, su coste será el mismo. Esto quiere decir que para cualquier nodo repetido, su coste será peor o igual que algún nodo anterior visitado o no, de manera que lo podremos descartar, ya que o lo hemos expandido ya o lo haremos próximamente.

El coste espacial de guardar los nodos cerrados para el tratamiento de repetidos es también $O(r^p)$. No hacer el tratamiento de nodos repetidos no supone ninguna ganancia, ya que tanto la cola de nodos abiertos como la de nodos repetidos crecen exponencialmente.

2.3 Búsqueda en profundidad prioritaria

La búsqueda en profundidad prioritaria intenta seguir un camino hasta la mayor profundidad posible, retrocediendo cuando acaba el camino y retomando la última posibilidad de elección disponible.

Para obtener este algoritmo solo hemos de instanciar la estructura que guarda los nodos abiertos a una pila en el algoritmo genérico. Esta estructura nos consigue que se visiten los nodos en el orden que hemos establecido.

El principal problema de este algoritmo es que no acaba si existe la posibilidad de que hayan caminos infinitos. Una variante de este algoritmo que evita este problema es el algoritmo de **profundidad limitada**, éste impone un límite máximo de profundidad que impone la longitud máxima de los caminos recorridos. Esta limitación garantiza que el algoritmo acaba, pero no garantiza la solución ya que ésta puede estar a mayor profundidad que el límite impuesto.

El algoritmo de profundidad limitada se puede ver en el algoritmo 2.1. La única diferencia en la implementación de esta variante con el algoritmo general es que se dejan de generar sucesores cuando se alcanza la profundidad límite.

Si nos fijamos en las propiedades que hemos escogido para clasificar los algoritmos:

- Completitud: El algoritmo encuentra una solución si se impone una profundidad límite y existe una solución dentro de ese límite.
- Complejidad temporal: Si tomamos como medida del coste el número de nodos explorados, el coste es exponencial respecto al factor de ramificación y la profundidad del límite de exploración $O(r^p)$.

Algoritmo 2.1 Algoritmo de profundidad limitada

Procedimiento: Búsqueda en profundidad limitada (límite: entero)

```

Est_abiertos.insertar(Estado inicial)
Actual ← Est_abiertos.primer()
mientras no es_final?(Actual) y no Est_abiertos.vacia?() hacer
  Est_abiertos.borrar_primer()
  Est_cerrados.insertar(Actual)
  si profundidad(Actual) ≤ límite entonces
    Hijos ← generar_sucesores (Actual)
    Hijos ← tratar_repetidos (Hijos, Est_cerrados, Est_abiertos)
    Est_abiertos.insertar(Hijos)
  sino
    Actual ← Est_abiertos.primer()
  fin
fin

```

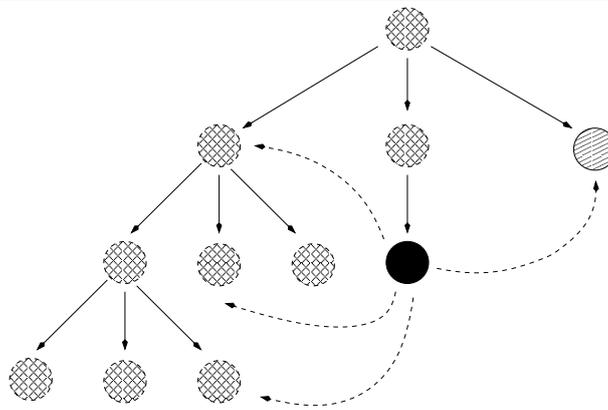


Figura 2.2: Tratamiento de repetidos en búsqueda en profundidad

- Complejidad espacial: El coste es lineal respecto al factor de ramificación y el límite de profundidad $O(rp)$. Si hacemos una implementación recursiva del algoritmo el coste es $O(p)$, ya que no nos hace falta generar todos los sucesores de un nodo, pudiéndolos tratar uno a uno. Si tratamos los nodos repetidos el coste espacial es igual que en anchura ya que tendremos que guardar todos los nodos cerrados.
- Optimalidad: No se garantiza que la solución sea óptima, la solución que se retornará será la primera en el orden de exploración.



Las implementaciones recursivas de los algoritmos permiten por lo general un gran ahorro de espacio. En este caso la búsqueda en profundidad se puede realizar recursivamente de manera natural (ver algoritmo 2.2). La recursividad permite que las alternativas queden almacenadas en la pila de ejecución como puntos de continuación del algoritmo sin necesidad de almacenar ninguna información. En este caso el ahorro de espacio es proporcional al factor de ramificación que en la práctica en problemas difíciles no es despreciable. Lo único que perdemos es la capacidad de buscar repetidos en los nodos pendientes que tenemos en el camino recorrido, pero habitualmente la limitación de memoria es una restricción bastante fuerte que impide solucionar problemas con longitudes de camino muy largas.

Algoritmo 2.2 Algoritmo de profundidad limitada recursivo

Función: Búsqueda en profundidad limitada recursiva (actual:nodo, limite: entero)

```

si profundidad(Actual) ≤ limite entonces
  para todo nodo ∈ generar_sucesores (Actual) hacer
    si es_final?(nodo) entonces
      | retorna (nodo)
    sino
      | resultado ← Búsqueda en profundidad limitada recursiva(nodo,limite)
      | si es_final?(resultado) entonces
      | | retorna (resultado)
      | fin
    fin
  fin
sino
  | retorna (∅)
fin

```

Respecto al tratamiento de nodos repetidos, si nos fijamos en la figura 2.2 podemos ver los diferentes casos con los que nos encontramos. El nodo de color negro sería el nodo generado actual, los nodos en cuadrícula son nodos cerrados y los nodos rayados son nodos abiertos. Si el nodo generado actual está repetido en niveles superiores (más cerca de la raíz), su coste será peor ya que su camino desde la raíz es más largo, si está al mismo nivel, su coste será el mismo. En estos dos casos podemos olvidarnos de este nodo.

En el caso de que el repetido corresponda a un nodo de profundidad superior, significa que hemos llegado al mismo estado por un camino más corto, de manera que deberemos mantenerlo y continuar su exploración, ya que nos permitirá llegar a mayor profundidad que antes.

En el caso de la búsqueda en profundidad, el tratamiento de nodos repetidos no es crucial ya que al tener un límite en profundidad los ciclos no llevan a caminos infinitos. No obstante, este caso se puede tratar comprobando los nodos en el camino actual ya que está completo en la estructura de nodos abiertos. Además, no tratando repetidos mantenemos el coste espacial lineal, lo cual es una gran ventaja.

El evitar tener que tratar repetidos y tener un coste espacial lineal supone una característica diferenciadora de hace muy ventajosa a la búsqueda en profundidad. Este algoritmo será capaz de obtener soluciones que se encuentren a gran profundidad.

En un problema concreto, el algoritmo de búsqueda en anchura tendrá una estructura de nodos abiertos de tamaño $O(r^p)$, lo que agotará rápidamente la memoria impidiendo continuar la búsqueda, pero solo habiendo alcanzado una profundidad de camino de p . En cambio el algoritmo en profundidad solo tendrá $O(rp)$ nodos abiertos al llegar a esa misma profundidad.

Esto quiere decir que en problemas difíciles la estrategia en profundidad será la única capaz de hallar una solución con un espacio de memoria limitado.

2.4 Búsqueda en profundidad iterativa

Este algoritmo intenta obtener las propiedades ventajosas de la búsqueda en profundidad y en anchura combinadas, es decir, un coste espacial lineal y asegurar que la solución será óptima respecto a la longitud del camino.

Para obtener esto lo que se hace es repetir sucesivamente el algoritmo de profundidad limitada,

Algoritmo 2.3 Algoritmo de profundidad iterativa (Iterative DeepeningD)

```

Procedimiento: Búsqueda en profundidad iterativa (limite: entero)
prof ← 1
Actual ← Estado inicial
mientras no es_final?(Actual) y prof < limite hacer
    Est_abiertos.inicializar()
    Est_abiertos.insertar(Estado inicial)
    Actual ← Est_abiertos.primer()
    mientras no es_final?(Actual) y no Est_abiertos.vacia?() hacer
        Est_abiertos.borrar_primer()
        Est_cerrados.insertar(Actual)
        si profundidad(Actual) ≤ prof entonces
            Hijos ← generar_sucesores (Actual)
            Hijos ← tratar_repetidos (Hijos, Est_cerrados, Est_abiertos)
            Est_abiertos.insertar(Hijos)
        sino
            Actual ← Est_abiertos.primer()
        fin
    fin
    prof ← prof+1
fin

```

aumentando a cada iteración la profundidad máxima a la que le permitimos llegar.

Este algoritmo obtendría el mismo efecto que el recorrido en anchura en cada iteración, ya que a cada paso recorremos un nivel más del espacio de búsqueda. El coste espacial será lineal ya que cada iteración es un recorrido en profundidad. Para que el algoritmo acabe en el caso de que no haya solución se puede imponer un límite máximo de profundidad en la búsqueda.

Aparentemente podría parecer que este algoritmo es más costoso que los anteriores al tener que repetir en cada iteración la búsqueda anterior, pero si pensamos en el número de nodos nuevos que recorremos a cada iteración, estos son siempre tantos como todos los que hemos recorrido en todas las iteraciones anteriores, por lo que las repeticiones suponen un factor constante respecto a los que recorreríamos haciendo solo la última iteración.

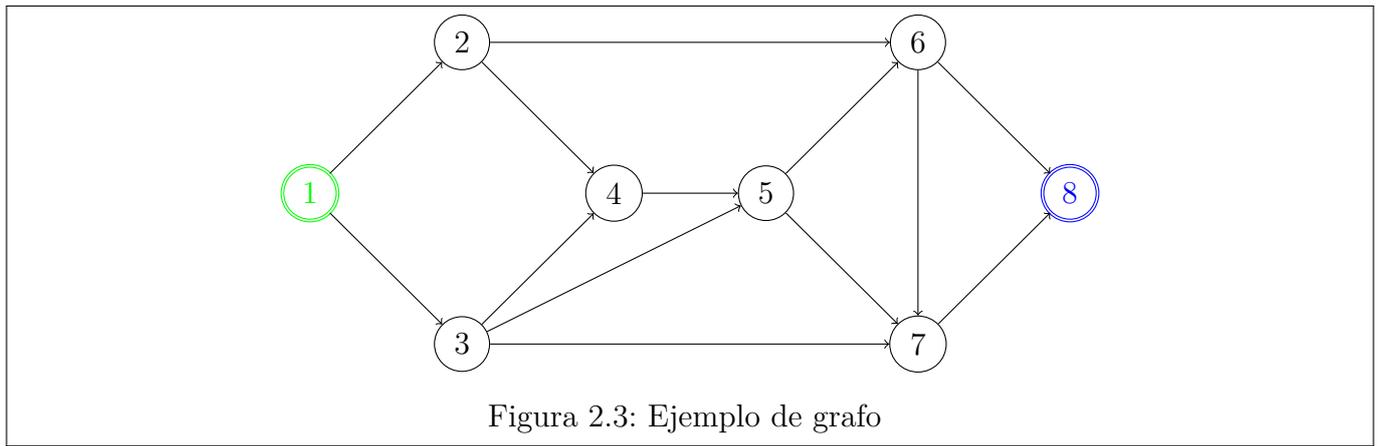
El algoritmo de profundidad iterativa se puede ver en el algoritmo [2.3](#).

Si nos fijamos en las propiedades que hemos escogido para clasificar los algoritmos:

- Completitud: El algoritmo siempre encontrará la 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 $O(r^p)$
- Complejidad espacial: Igual que en la búsqueda en profundidad
- Optimalidad: La solución es óptima igual que en la búsqueda en anchura

Igual que en el caso del algoritmo en profundidad, el tratar repetidos acaba con todas las ventajas espaciales del algoritmo, por lo que es aconsejable no hacerlo. Como máximo se puede utilizar la estructura de nodos abiertos para detectar bucles en el camino actual.

Si el algoritmo en profundidad limitada es capaz de encontrar soluciones a mayor profundidad, este algoritmo además nos garantizará que la solución será óptima. Por lo tanto este es el algoritmo más ventajoso de los tres.



2.5 Ejemplos

En esta sección vamos a ver ejemplos de ejecución de los algoritmos vistos en estos capítulos. En la figura 2.3 podemos ver el grafo que vamos a utilizar en los siguientes ejemplos. Este grafo tiene como nodo inicial el 1 y como nodo final el 8, todos los caminos son dirigidos y tienen el mismo coste. Utilizaremos los algoritmos de búsqueda ciega para encontrar un camino entre estos dos nodos.

Supondremos que hacemos un tratamiento de los nodos repetidos durante la ejecución del algoritmo. Hay que tener en cuenta también que para poder recuperar el camino, los nodos deben tener un enlace al padre que los ha generado.

Búsqueda en anchura

1. Este algoritmo comenzaría encolando el nodo inicial (nodo 1).
2. Los nodos sucesores del primer nodo son los nodos 2 y 3, que pasarían a ser encolados
3. El siguiente nodo extraído de la cola sería el 2, que tiene como sucesores el 4 y el 6.
4. El siguiente nodo extraído de la cola sería el 3, que tiene como sucesores el 4, el 5 y el 7. Como el 4 está repetido no se encolaría al ser de la misma profundidad que el nodo repetido.
5. El siguiente nodo extraído de la cola sería el 4, que tiene como sucesor el 5. Como el 4 está repetido no se encolaría al tener profundidad mayor que el nodo repetido.
6. El siguiente nodo extraído de la cola sería el 6, que tiene como sucesores el 7 y el 8. Como el 7 está repetido no se encolaría al tener profundidad mayor que el nodo repetido.
7. El siguiente nodo extraído de la cola sería el 5, que tiene como sucesores el 6 y el 7. Como los dos nodos corresponden a nodos repetidos con profundidad menor, no se encolarían.
8. El siguiente nodo extraído de la cola sería el 7, que tiene como sucesor el 8. Como el 8 está repetido no se encolaría al tener la misma profundidad que el nodo repetido.
9. El siguiente nodo extraído de la cola sería el 8 que es el nodo solución y por tanto acabaría la ejecución.

En la figura 2.4 se puede ver el árbol de búsqueda que se genera. En el caso de los nodos cerrados necesitamos una estructura adicional para controlar los nodos repetidos. resultado)

Búsqueda en profundidad

1. Este algoritmo comenzaría empilando el nodo inicial (nodo 1).

2. Los nodos sucesores del primer nodo son los nodos 2 y 3, que pasarían a ser empilados
3. El siguiente nodo extraído de la pila sería el 2, que tiene como sucesores el 4 y el 6.
4. El siguiente nodo extraído de la pila sería el 4, que tiene como sucesor el 5.
5. El siguiente nodo extraído de la pila sería el 5, que tiene como sucesores el 6 y el 7. Como el nodo 6 está en la pila en con nivel superior descartaríamos este nodo.
6. El siguiente nodo extraído de la pila sería el 7, que tiene como sucesor el 8.
7. El siguiente nodo extraído de la pila sería el 8 que es el nodo solución y por tanto acabaría la ejecución.

En la figura 2.5 se puede ver el árbol de búsqueda que se genera. Fijaos que en este caso el camino es más largo que el encontrado con el algoritmo anterior. De hecho es el primer camino solución que se encuentra en la exploración según la ordenación que se hace de los nodos. Para el control de repetidos, si se quiere ahorrar espacio, pueden solo tenerse en cuenta los nodos que hay en la pila como se ha hecho en el ejemplo, esto evitará que podamos entrar en bucle y que tengamos que tener una estructura para los nodos cerrados.

Búsqueda en profundidad iterativa

1. Primera iteración (profundidad 1)
 - 1.1 Se comenzaría empilando el nodo inicial (nodo 1).
 - 1.2 Al extraer este nodo de la pila habríamos visitado todos los caminos de profundidad 1 y con eso acabaríamos la iteración.
2. Segunda Iteración (profundidad 2)
 - 2.1 Se comenzaría empilando el nodo inicial (nodo 1).
 - 2.2 Los nodos sucesores del primer nodo son los nodos 2 y 3, que pasarían a ser empilados.
 - 2.3 El siguiente nodo extraído de la pila sería el 2, los nodos sucesores estarían a mayor profundidad que el límite actual, no los empilaríamos.
 - 2.3 El siguiente nodo extraído de la pila sería el 3, los nodos sucesores estarían a mayor profundidad que el límite actual, no los empilaríamos. Al extraer este nodo de la pila habríamos visitado todos los caminos de profundidad 2 y con eso acabaríamos la iteración.
3. Tercera Iteración (profundidad 3)
 - 3.1 Se comenzaría empilando el nodo inicial (nodo 1).
 - 3.2 Los nodos sucesores del primer nodo son los nodos 2 y 3, que pasarían a ser empilados.
 - 3.3 El siguiente nodo extraído de la pila sería el 2, que tiene como sucesores el 4 y el 6.
 - 3.4 El siguiente nodo extraído de la pila sería el 4, los nodos sucesores estarían a mayor profundidad que el límite actual, no los empilaríamos.
 - 3.5 El siguiente nodo extraído de la pila sería el 6, los nodos sucesores estarían a mayor profundidad que el límite actual, no los empilaríamos.
 - 3.6 El siguiente nodo extraído de la pila sería el 3, que tiene como sucesores el 4, el 5 y el 7.
 - 3.7 El siguiente nodo extraído de la pila sería el 4, los nodos sucesores estarían a mayor profundidad que el límite actual, no los empilaríamos.

- 3.8 El siguiente nodo extraído de la pila sería el 5, los nodos sucesores estarían a mayor profundidad que el límite actual, no los empilaríamos.
- 3.4 El siguiente nodo extraído de la pila sería el 7, los nodos sucesores estarían a mayor profundidad que el límite actual, no los empilaríamos. Al extraer este nodo de la pila habríamos visitado todos los caminos de profundidad 3 y con eso acabaríamos la iteración.

4. Cuarta Iteración (profundidad 4)

- 4.1 Se comenzaría empilando el nodo inicial (nodo 1).
- 4.2 Los nodos sucesores del primer nodo son los nodos 2 y 3, que pasarían a ser empilados.
- 4.3 El siguiente nodo extraído de la pila sería el 2, que tiene como sucesores el 4 y el 6.
- 4.4 El siguiente nodo extraído de la pila sería el 4, que tiene como sucesor el 5.
- 4.5 El siguiente nodo extraído de la pila sería el 5, los nodos sucesores estarían a mayor profundidad que el límite actual, no los empilaríamos.
- 4.6 El siguiente nodo extraído de la pila sería el 6, que tiene como sucesores el 7 y el 8.
- 4.7 El siguiente nodo extraído de la pila sería el 7, los nodos sucesores estarían a mayor profundidad que el límite actual, no los empilaríamos.
- 4.8 El siguiente nodo extraído de la pila sería el 8 que es el nodo solución y por tanto acabaría la ejecución.

En la figura 2.6 se puede ver el árbol de búsqueda que se genera. En este caso encontramos el camino mas corto, ya que este algoritmo explora los caminos en orden de longitud como la búsqueda en anchura. Para mantener la ganancia en espacio en este caso no se han guardado los nodos cerrados, es por ello que no se ha tratado en nodo 4 en el paso 3.7 como repetido.

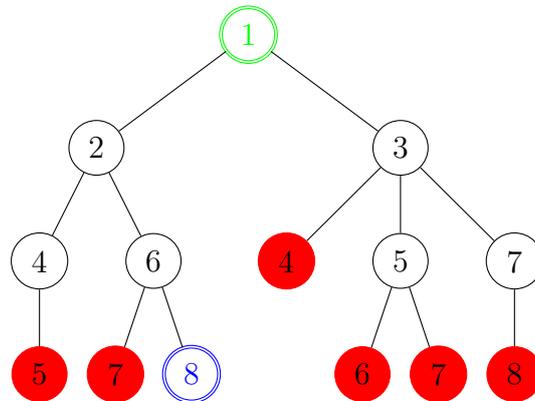


Figura 2.4: Ejecución del algoritmo de búsqueda en anchura

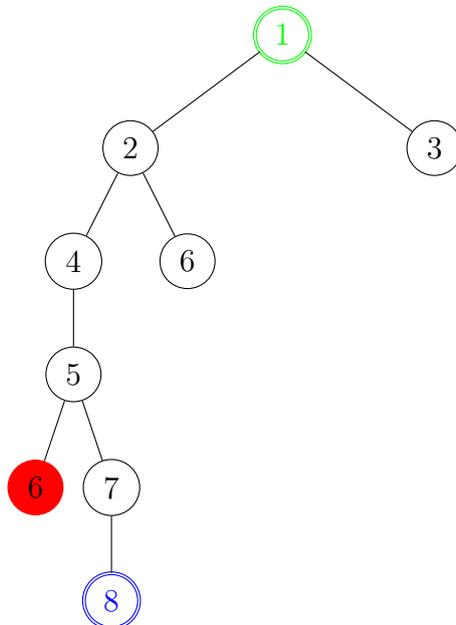


Figura 2.5: Ejecución del algoritmo de búsqueda en profundidad

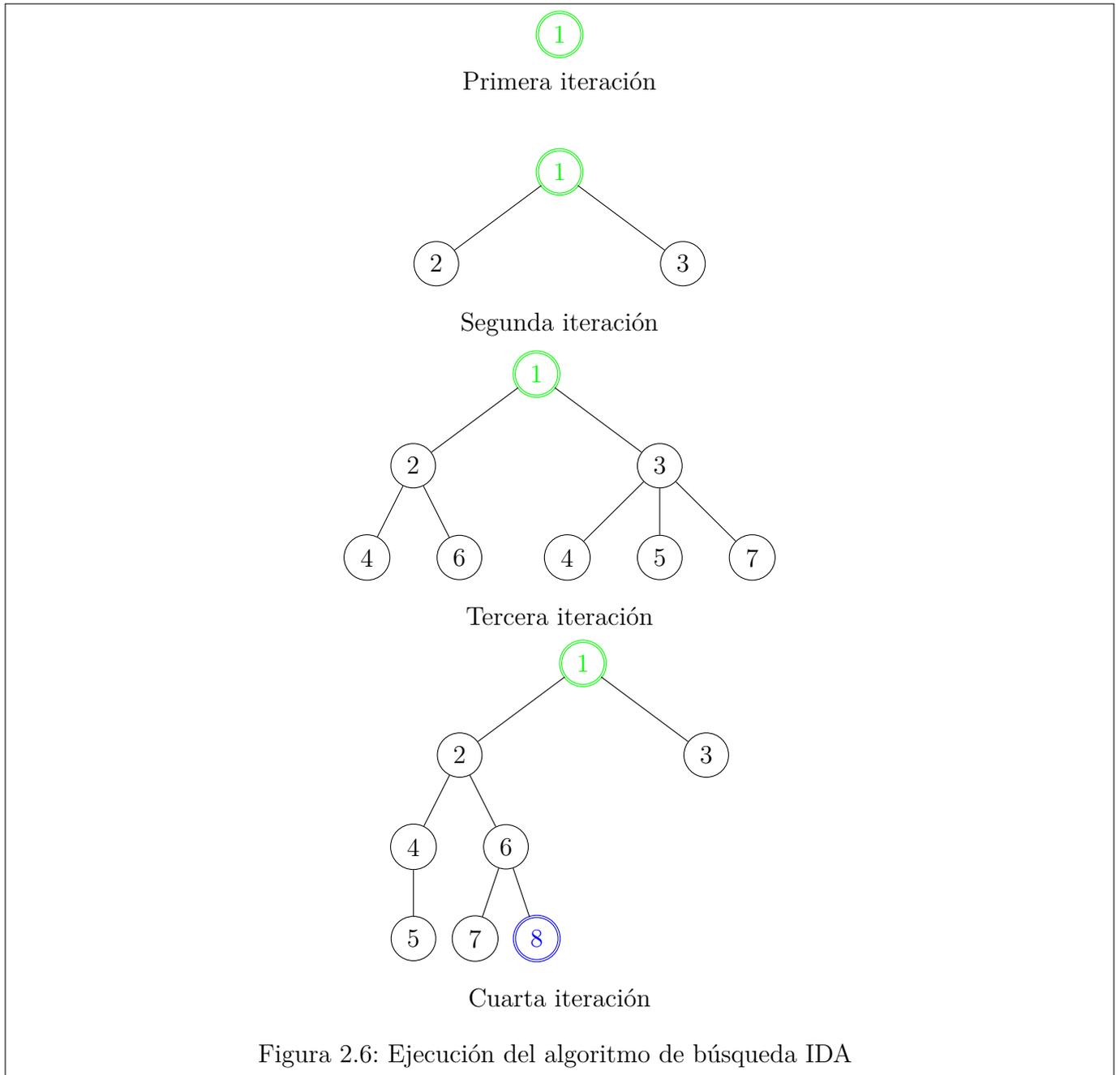


Figura 2.6: Ejecución del algoritmo de búsqueda IDA

3.1 El conocimiento importa

Es evidente que los algoritmos de búsqueda no informada serán incapaces de encontrar soluciones en problemas en los que el tamaño del espacio de búsqueda sea grande. Todos estos algoritmos tienen un coste temporal que es una función exponencial del tamaño de la entrada, por lo tanto el tiempo para encontrar la mejor solución a un problema no es asumible en problemas reales.

La manera de conseguir reducir este tiempo de búsqueda a algo más razonable es intentar hacer intervenir conocimiento sobre el problema que queremos resolver dentro del funcionamiento del algoritmo de búsqueda.

El problema que tendremos 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.

En este capítulo nos centraremos en los algoritmos que buscan la solución óptima. Este objetivo solo es abordable hasta cierto tamaño de problemas, existirá todo un conjunto de problemas en los que la búsqueda del óptimo será imposible por el tamaño de su espacio de búsqueda o por la imposibilidad de encontrar información que ayude en la búsqueda.

3.2 El óptimo está en el camino

Para poder plantear la búsqueda del óptimo siempre hemos que tener alguna medida del coste de obtener una solución. Este coste lo mediremos sobre el camino que nos lleva desde el estado inicial del problema hasta el estado final. No todos los problemas se pueden plantear de esta forma tal y como veremos en el próximo capítulo.

Por lo tanto, supondremos que podemos plantear nuestra búsqueda como la búsqueda de un camino y que de alguna manera somos capaces de saber o estimar cuál es la longitud o coste de éste. 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.

Los algoritmos que veremos utilizarán el cálculo del coste de los caminos explorados para saber que nodos merece la pena explorar antes. De esta manera, perderemos la sistematicidad de los algoritmos de búsqueda no informada, y el orden de visita de los estados de búsqueda no vendrá determinado por su posición en el grafo de búsqueda, sino por su coste.

Dado que el grafo va siendo generado a medida que lo exploramos, podemos ver que tenemos dos elementos que intervienen en el coste del camino hasta la solución que buscamos. En primer lugar, tendremos el coste del camino recorrido, que podremos calcular simplemente sumando los costes de los operadores aplicados desde el estado inicial hasta el nodo actual. En segundo lugar, tendremos el coste más difícil, que es el del camino que nos queda por recorrer hasta el estado final. Dado que lo desconocemos, tendremos que utilizar el conocimiento del que disponemos del problema para obtener una aproximación.

Evidentemente, la calidad de ese conocimiento que nos permite predecir el coste futuro, hará más o menos exitosa nuestra búsqueda. Si nuestro conocimiento fuera perfecto, podríamos dirigirnos rápidamente hacia el objetivo descartando todos los caminos de mayor coste, en este extremo podríamos

Algoritmo 3.1 Algoritmo Greedy Best First

```

Algoritmo: Greedy Best First
Est_abiertos.insertar(Estado inicial)
Actual ← Est_abiertos.primer()
mientras no es_final?(Actual) y no Est_abiertos.vacía?() hacer
    Est_abiertos.borrar_primer()
    Est_cerrados.insertar(Actual)
    hijos ← generar_sucesores (Actual)
    hijos ← tratar_repetidos (Hijos, Est_cerrados, Est_abiertos)
    Est_abiertos.insertar(Hijos)
    Actual ← Est_abiertos.primer()
fin

```

encontrar nuestra solución en tiempo lineal. En el otro extremo, si estuviéramos en la total ignorancia, tendríamos que explorar muchos caminos antes de hallar la solución óptima, todos en el peor caso. Esta última situación es en la que se encuentra la búsqueda no informada y desgraciadamente la primera situación es rara.

Quedará pues como labor fundamental en cada problema, obtener una función que nos haga el cálculo del coste futuro desde un estado al estado solución. Cuanto más podamos ajustarlo al coste real, mejor funcionarán los algoritmos que veremos a continuación.

3.3 Tú primero y tú después

El fundamento de los algoritmos de búsqueda heurística será el cómo elegir qué nodo explorar primero, para ello podemos utilizar diferentes estrategias que nos darán diferentes propiedades.

Una primera estrategia que se nos puede ocurrir es utilizar la estimación del coste futuro para decidir qué nodos explorar primero. De esta manera supondremos que los nodos que aparentemente están más cerca de la solución formarán parte del camino hasta la solución óptima y, por lo tanto, la encontraremos antes si los exploramos en primer lugar.

Esta estrategia se traduce en el algoritmo *primero el mejor avaricioso* (**greedy best first**). La única diferencia respecto a los algoritmos que hemos visto es el utilizar como estructura para almacenar los nodos abiertos una cola con prioridad. La prioridad de los nodos la marca la estimación del coste del camino del nodo hasta el nodo solución. El algoritmo 3.1 es su implementación.

El explorar antes los nodos más cercanos a la solución probablemente nos hará encontrarla antes, pero el no tener en cuenta el coste del camino recorrido hace que no se garantice la solución óptima, ya que a pesar de guiarnos hacia el nodo aparentemente más cercano a la solución, estamos ignorando el coste del camino completo.

3.4 El algoritmo A*

Dado que nuestro objetivo no es solo llegar lo más rápidamente a la solución, sino encontrar la de menor coste tendremos que tener en cuenta el coste de todo el camino y no solo el camino por recorrer. Para poder introducir el siguiente algoritmo y establecer sus propiedades tenemos primero que dar una serie de definiciones.

Denominaremos el **coste de un arco** entre dos nodos n_i y n_j al coste del operador que nos

Algoritmo 3.2 Algoritmo A***Algoritmo:** A*

Est_abiertos.insertar(Estado inicial)

Actual \leftarrow Est_abiertos.primerero()**mientras no es_final?(Actual) y no Est_abiertos.vacía?() hacer**

Est_abiertos.borrar_primerero()

Est_cerrados.insertar(Actual)

 hijos \leftarrow generar_sucesores (Actual) hijos \leftarrow tratar_repetidos (Hijos, Est_cerrados, Est_abiertos)

Est_abiertos.insertar(Hijos)

 Actual \leftarrow Est_abiertos.primerero()**fin**

permite pasar de un nodo al otro, y lo denotaremos como $c(n_i, n_j)$. Este coste siempre será positivo.

Denominaremos el **coste de un camino** entre dos nodos n_i y n_j a la suma de los costes de todos los arcos que llevan desde un nodo al otro y lo denotaremos como

$$C(n_i, n_j) = \sum_{x=i}^{j-1} c(n_x, n_{x+1})$$

Denominaremos el **coste del camino mínimo** entre dos nodos n_i y n_j al camino de menor coste de entre los que llevan desde un nodo al otro y lo denotaremos como

$$K(n_i, n_j) = \min_{k=1}^l C_k(n_i, n_j)$$

Si n_j es un nodo terminal, llamaremos $h^*(n_i)$ a $K(n_i, n_j)$, es decir, el coste del camino mínimo desde un estado cualquiera a un estado solución.

Si n_i es un nodo inicial, llamaremos $g^*(n_j)$ a $K(n_i, n_j)$, 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 mas el coste del nodo hasta el nodo final:

$$f^*(n) = g^*(n) + h^*(n)$$

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 le daremos el nombre de **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 ese coste es conocido ya que ese camino lo hemos recorrido en nuestra exploración. De esta manera tendremos una estimación del coste del camino mínimo que pasa por cierto nodo:

$$f(n) = g(n) + h(n)$$

Será este valor el que utilicemos para decidir en nuestro algoritmo de búsqueda cuál es el siguiente nodo a explorar de entre todos los nodos abiertos disponibles. Para realizar esa búsqueda utilizaremos el algoritmo que denominaremos A* (algoritmo 3.2).

Como se observará, el algoritmo es el mismo que el **primero el mejor avaricioso**, lo único que cambia es que ahora la ordenación de los nodos se realiza utilizando el valor de f . Como criterio de ordenación, consideraremos que a igual valor de f los nodos con h más pequeña se explorarán antes

(simplemente porque si la h es más pequeña es que son nodos mas cerca de la solución), a igual h se consideraran en el orden en el que se introdujeron en la cola.

Nunca está de más el remarcar que el algoritmo solo acaba cuando se **extrae** una solución de la cola. Es posible que en cierto momento ya haya en la estructura de nodos abiertos nodos solución, pero hasta que no se hayan explorado los nodos por delante de ellos, no podemos asegurar que realmente sean soluciones buenas. Siempre hay que tener en mente que los nodos están ordenados por el coste estimado del camino total, si la estimación es menor es que podrían pertenecer a un camino con una solución mejor.

Hay que considerar que el coste de este algoritmo es también $O(r^p)$ en el caso peor. Si por ejemplo, la función $h(n)$ fuera siempre 0, el algoritmo se comportaría como una búsqueda en anchura gobernada por el coste del camino recorrido.

Lo que hace que este algoritmo pueda tener un coste temporal inferior es la bondad de la función h . De hecho, podemos interpretar las funciones g y h como las que gobiernan el comportamiento en anchura o profundidad del algoritmo.

Cuanto más cercana al coste real sea h , mayor será el comportamiento en profundidad del algoritmo, pues los nodos que aparentemente están más cerca de la solución se explorarán antes. En el momento en el que esa información deje de ser fiable, el coste del camino ya explorado hará que otros nodos menos profundos tengan un coste total mejor y, por lo tanto, se abrirá la búsqueda en anchura.

Si pensamos un poco en el coste espacial que tendrá el algoritmo podemos observar que como éste puede comportarse como un algoritmo de búsqueda en anchura, el coste espacial en el caso peor también será $O(r^p)$. Igual nos pasa con el coste temporal, no obstante, si la función heurística es suficientemente buena no llegaremos a necesitar tanto tiempo, ni espacio antes de llegar a la solución. Un resultado a tener en cuenta es que el coste del algoritmo A^* crece exponencialmente a no ser que se cumpla que:

$$|h(n) - h^*(n)| < O(\log(h^*(n)))$$

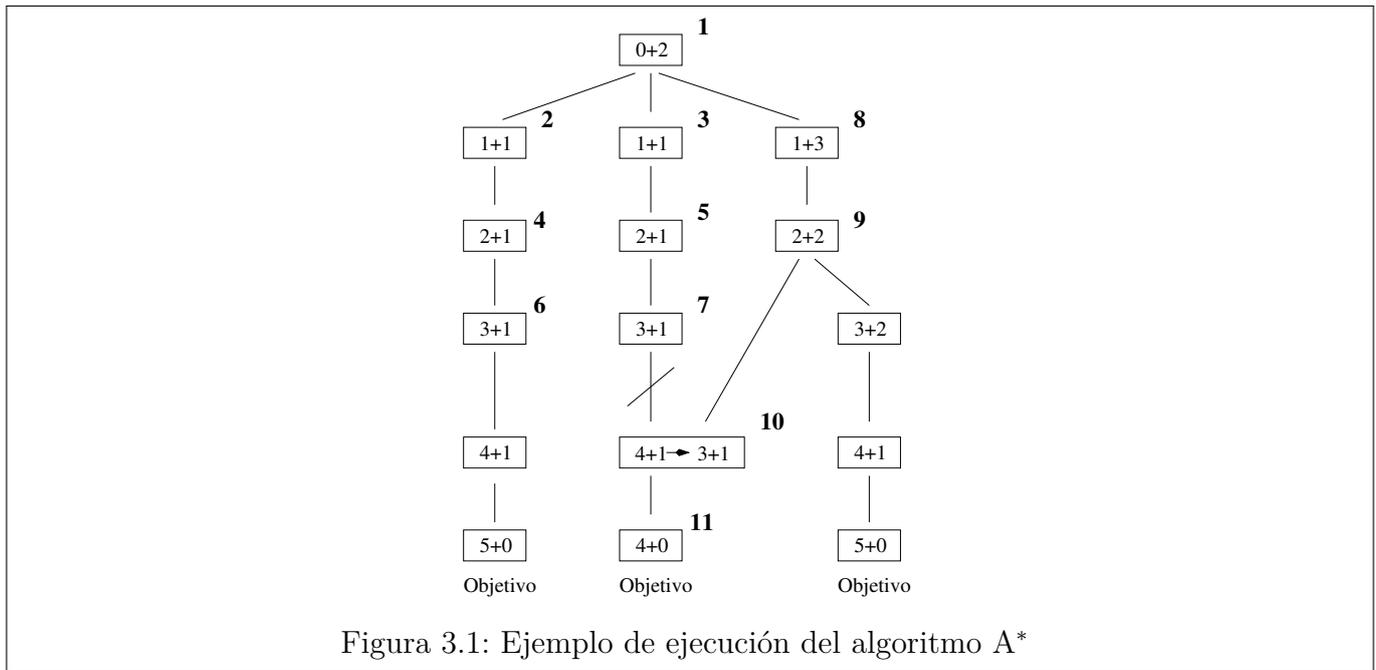
Esto pondrá el límite respecto a cuando podemos obtener una solución óptima para un problema. Si no podemos encontrar una función heurística suficientemente buena, deberemos usar algoritmos que no busquen el óptimo, pero que nos garanticen que nos acercaremos lo suficiente a él para obtener una buena solución.

El tratamiento de nodos repetidos en este algoritmo se realiza de la siguiente forma:

- Si es un nodo repetido que está en la estructura de nodos abiertos
 - Si su coste es menor substituímos 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 nodo repetido que está en la estructura de nodos 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 reabrirán si hace falta
 - Si su coste es mayor o igual nos olvidamos del nodo

Mas adelante veremos que si la función h cumple ciertas condiciones podemos evitar el tratamiento de repetidos.

A continuación podemos ver un pequeño ejemplo de ejecución del algoritmo A^*



Ejemplo 3.1 Si consideramos el árbol de búsqueda de la figura 3.1, donde en cada nodo tenemos descompuesto el coste en g y h y marcado el orden en el que el algoritmo ha visitado cada nodo.

La numeración en los nodos indica el orden en que el algoritmo A^* los ha expandido (y por lo tanto el orden en el que han sido extraídos de la cola).

Podemos observar también que hay un nodo abierto que es revisitado por otro camino con un coste inferior que substituye al encontrado con anterioridad.

3.5 Pero, ¿encontraré el óptimo?

Hasta ahora no hemos hablado para nada sobre como garantizar la optimalidad de la solución. Hemos visto que en el caso degenerado tenemos una búsqueda en anchura guiada por el coste del camino explorado, eso nos debería garantizar el óptimo en este caso. Pero como hemos comentado, la función h nos permite obtener un comportamiento de búsqueda en profundidad y sabemos que en este caso no se nos garantiza el óptimo.

El saber si encontraremos o no el óptimo mediante el algoritmo A^* recae totalmente en las propiedades que cumple la función heurística, si esta cumple ciertos criterios sabremos que encontraremos la solución óptima, si no los cumple, no lo podremos saber.

3.5.1 Admisibilidad

La propiedad clave que nos garantizará el hallar la solución óptima es la que denominaremos **admisibilidad**. Diremos que una función heurística h es admisible siempre que se cumpla que su valor en cada nodo sea menor o igual que el valor del coste real del camino que nos falta por recorrer hasta la solución:

$$\forall n \quad 0 \leq h(n) \leq h^*(n)$$

Esto quiere decir que la función heurística ha de ser un estimador *optimista* del coste que falta para llegar a la solución.

Ejemplo 3.2 En la figura 3.2 podemos ver en este ejemplo dos grafos de búsqueda, uno con una función admisible y otra que no lo es.

En este primer caso, la función heurística siempre es admisible, pero en la primera rama el coste es más pequeño que el real, por lo que pierde cierto tiempo explorándola (el efecto en profundidad que hemos comentado), pero al final el algoritmo pasa a la segunda rama hallando la solución.

En el segundo caso el algoritmo acabaría al encontrar la solución en G, a pesar de que haya una solución con un coste más pequeño en H. Esto es así porque el coste estimado que da en el nodo D es superior al real y eso le relega en la cola de nodos abiertos.

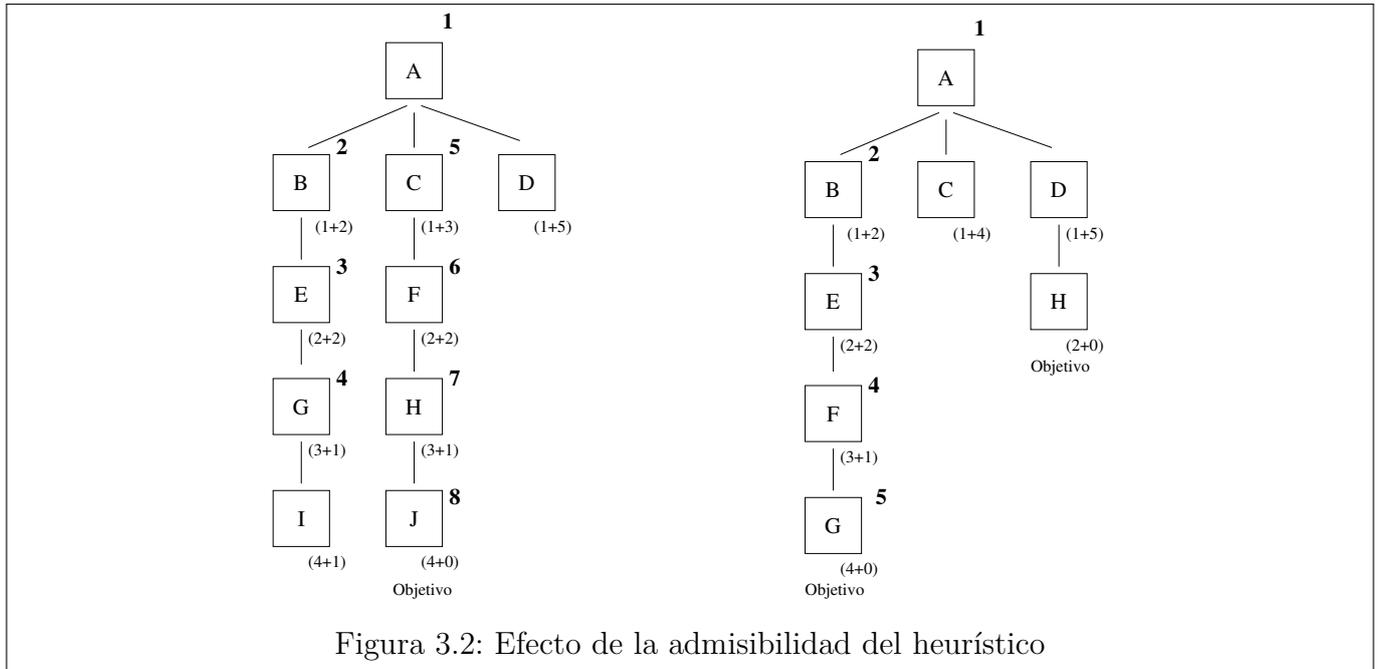


Figura 3.2: Efecto de la admisibilidad del heurístico

Esta propiedad implica que, si podemos demostrar que la función de estimación h que utilizamos en nuestro problema la cumple, siempre encontraremos una solución óptima. El problema radica en hacer esa demostración, pues cada problema es diferente. Por ello, no podemos dar un método general para demostrar la admisibilidad de una función heurística.

Lo que sí podemos establecer, es que para demostrar que una función heurística **no** es admisible basta con encontrar un nodo que incumpla la propiedad. Esto se puede observar simplemente comprobando si alguno de los nodos que están en el camino solución tiene un coste mayor que el real, pues solo disponemos del coste real para los nodos de ese camino.

Hay que remarcar también que la propiedad de admisibilidad es una propiedad de la función heurística, y es independiente del algoritmo que la utiliza, por lo que si la función es admisible cualquier algoritmo de los que veamos que la utilice nos hallará la solución óptima.

Para una discusión sobre técnicas para construir heurísticos admisibles se puede consultar el capítulo 4, sección 4.2, del libro “Inteligencia Artificial: Un enfoque moderno” de S. Russell y P. Norvig .

3.5.2 Consistencia

Podemos definir la propiedad de consistencia como una extensión de la propiedad de admisibilidad. Si tenemos el coste $h^*(n_i)$ y el coste $h^*(n_j)$ y el coste óptimo para ir de n_i a n_j , o sea $K(n_i, n_j)$, se ha de cumplir la desigualdad triangular:

$$h^*(n_i) \leq h^*(n_j) + K(n_i, n_j)$$

La condición de consistencia exige pedir lo mismo al estimador h :

$$h(n_i) - h(n_j) \leq K(n_i, n_j)$$

Es decir, que 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 hemos llegado 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 camino alternativo con un coste menor y esto hace que el tratamiento de nodos cerrados duplicados sea innecesario, lo que nos ahorra tener que guardarlos.

Habitualmente las funciones heurísticas admisibles suelen ser consistentes y, de hecho, hay que esforzarse bastante para conseguir que no sea así.

3.5.3 Heurístico más informado

Otra propiedad interesante es la que nos permite comparar heurísticos entre si y nos permite saber cuál de ellos hará que A^* encuentre más rápido una solución óptima. Diremos que el heurístico h_1 es más informado que el heurístico h_2 si se cumple la propiedad:

$$\forall n \quad 0 \leq h_2(n) < h_1(n) \leq h^*(n)$$

Se ha de observar que esta propiedad incluye que los dos heurísticos sean admisibles.

Si un heurístico es más informado que otro seguro que A^* expandirá menos nodos durante la búsqueda, ya que su comportamiento será más en profundidad y habrá ciertos nodos que no se explorarán.

Esto nos podría hacer pensar que siempre tenemos que escoger el heurístico más informado. Hemos de pensar también que la función heurística tiene un tiempo de cálculo que afecta al tiempo de cada iteración. 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 (ϵ -admisibilidad) que garantizan una solución dentro de un rango del valor del coste óptimo.

3.6 Mi memoria se acaba

El algoritmo A^* tiene sus limitaciones de espacio impuestas en primer lugar por poder acabar degenerando en una búsqueda en anchura si la función heurística no es demasiado buena. Además, si la solución del problema está a mucha profundidad o el tamaño del espacio de búsqueda es muy grande, no hace falta mucho para llegar a necesidades de espacio prohibitivas. Esto nos lleva a plantearnos algoritmos alternativos con menores necesidades de espacio.

Algoritmo 3.3 Algoritmo IDA*

```

Algoritmo: IDA* (limite entero)
prof ←  $f(\text{Estado inicial})$ 
Actual ← Estado inicial
mientras no es_final?(Actual) y  $prof < \text{limite}$  hacer
  Est_abiertos.inicializa()
  Est_abiertos.insertar(Estado inicial)
  Actual ← Est_abiertos.primer()
  mientras no es_final?(Actual) y no Est_abiertos.vacia?() hacer
    Est_abiertos.borrar_primer()
    Est_cerrados.insertar(Actual)
    Hijos ← generar_sucesores (Actual, prof)
    Hijos ← tratar_repetidos (Hijos, Est_cerrados, Est_abiertos)
    Est_abiertos.insertar(Hijos)
    Actual ← Est_abiertos.primer()
  prof ← prof+1

```

3.6.1 El algoritmo IDA*

La primera solución viene de la mano del algoritmo en profundidad iterativa que hemos visto en el capítulo anterior. En este caso lo replanteamos para utilizar la función f como el valor que controla la profundidad a la que llegamos en cada iteración.

Esto quiere decir que hacemos la búsqueda imponiendo un límite al coste del camino que queremos hallar (en la primera iteración, la f del nodo raíz), explorando en profundidad todos los nodos con f igual o inferior a ese límite y reiniciando la búsqueda con un coste mayor si no encontramos la solución en la iteración actual. Es el algoritmo 3.3.

La función `generar_sucesores` solo retorna los nodos con un coste inferior o igual al de la iteración actual.



Este algoritmo admite varias optimizaciones, como no aumentar el coste de uno en uno, sino averiguar cual es el coste más pequeño de los nodos no expandidos en la iteración actual y usarlo en la siguiente iteración. El bucle interior es el que realiza la búsqueda en profundidad limitada y también se podría optimizar utilizando una implementación recursiva.

Este algoritmo, al necesitar solo una cantidad de espacio lineal, permite hallar soluciones a más profundidad. El precio que hemos de pagar es el de las reexpansiones de nodos ya visitados.

Este precio extra dependerá de la conectividad del grafo del espacio de estados, si existen muchos ciclos este puede ser relativamente alto ya que a cada iteración la efectividad de la exploración real se reduce con el número de nodos repetidos que aparecen.

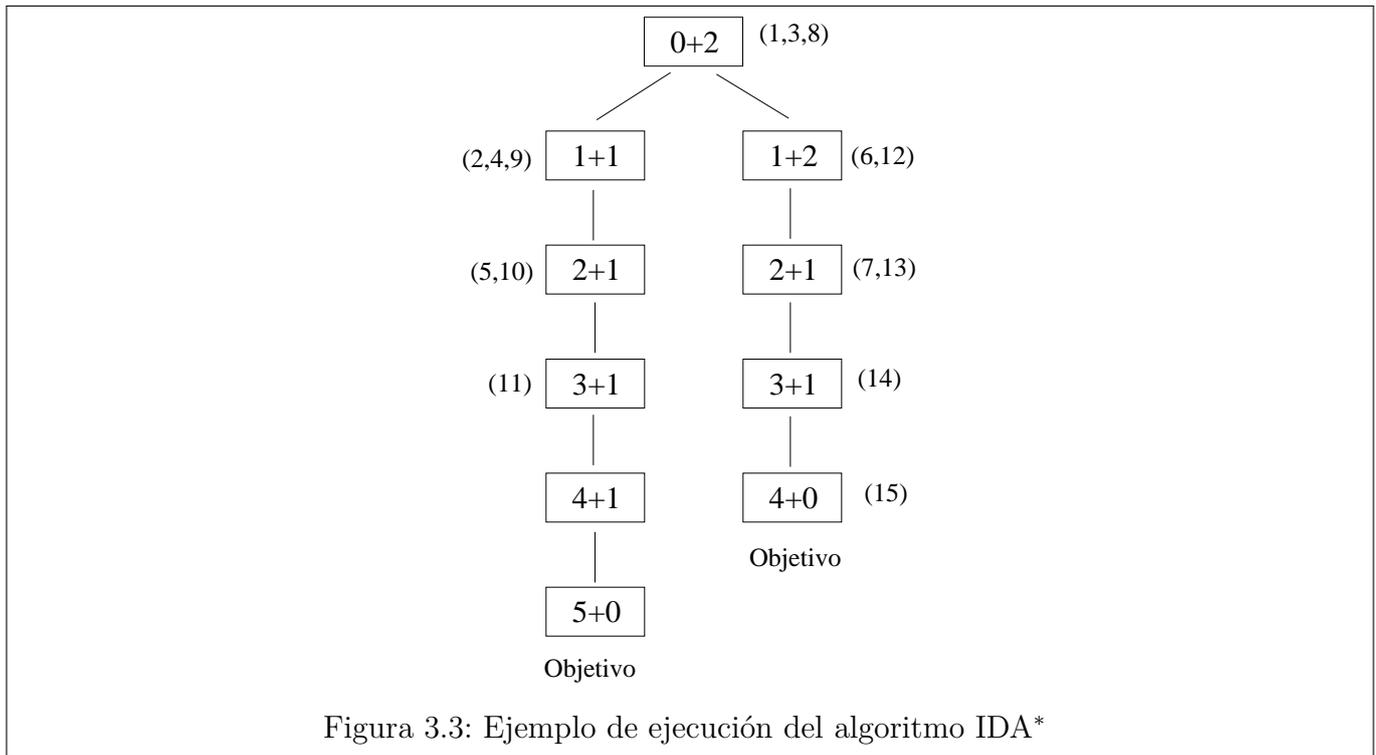


Figura 3.3: Ejemplo de ejecución del algoritmo IDA*



Si consideramos el cociente entre los nodos nuevos que se expanden en un nivel y los nodos que se han reexpandido, podemos ver que este tiende a 0 al aumentar el factor de ramificación, por lo que el trabajo de reexpandir los nodos es despreciable cuando el problema es complejo. Siendo r el factor de ramificación del problema, si consideramos que r^{n+1} será en número de nodos que visitaremos en el nivel $n + 1$ y $\sum_{i=1}^n r^i$ es la suma de nodos que revisitamos en ese nivel:

$$\frac{\sum_{i=1}^n r^i}{r^{n+1}} = \frac{r + r^2 + r^3 + \dots + r^n}{r^{n+1}} = \frac{1 + r + r^2 + r^3 + \dots + r^{n-1}}{r^n} = \frac{1}{r^n} + \frac{1}{r^{n-1}} + \dots + \frac{1}{r} = \frac{1}{r-1}$$

Ejemplo 3.3 Si consideramos el árbol de búsqueda de la figura 3.3, donde en cada nodo tenemos descompuesto el coste en g y h y marcado el orden en el que el algoritmo ha visitado cada nodo.

La numeración en los nodos indica el orden en que el algoritmo IDA* los ha expandido. Podemos observar que los nodos se reexpanden varias veces siguiendo la longitud del camino estimado impuesta en cada iteración del algoritmo-

3.6.2 Otras alternativas

Las reexpansiones del IDA* pueden llegar a ser un problema, e incrementar bastante el tiempo de búsqueda. Estas reexpansiones se deben fundamentalmente a que estamos imponiendo unas restricciones de espacio bastantes severas. Si relajáramos esta restricción manteniéndola en límites razonables podríamos guardar información suficiente para no tener que realizar tantas reexpansiones.

Una primera alternativa es el algoritmo **primero el mejor recursivo** (*recursive best first*). El elemento clave es la implementación recursiva, que permite que el coste espacial se mantenga lineal ($O(rp)$) al no tener que guardar mas que los nodos que pertenecen al camino actual y sus hermanos en cada nivel.

Este algoritmo intenta avanzar siempre por la rama más prometedora (según la función heurística f) hasta que alguno de los nodos alternativos del camino tiene un coste mejor. En este momento el

Algoritmo 3.4 Algoritmo Best First Recursivo

```

Procedimiento: BFS-recursivo (nodo,c_alternativo,ref nuevo_coste,ref solucion)
si es_solucion?(nodo) entonces
  | solucion.añadir(nodo)
sino
  | sucesores ← generar_sucesores (nodo)
  | si sucesores.vacio?() entonces
  | | nuevo_coste ←  $+\infty$ ; solucion.vacio()
  | sino
  | | fin ← falso
  | | mientras no fin hacer
  | | | mejor ← sucesores.mejor_nodo()
  | | | si mejor.coste() > c_alternativo entonces
  | | | | fin ← cierto; solucion.vacio(); nuevo_coste ← mejor.coste()
  | | | sino
  | | | | segundo ← sucesores.segundo_mejor_nodo()
  | | | | BFS-recursivo(mejor,min(c_alternativo,segundo.coste()),nuevo_coste, solucion)
  | | | | si solucion.vacio?() entonces
  | | | | | mejor.coste(nuevo_coste)
  | | | | sino
  | | | | | solucion.añadir(mejor); fin ← cierto

```

camino del nodo actual es olvidado, pero modificando la estimación de los ascendientes con la del nodo más profundo al que se ha llegado. Se podría decir que con cada reexpansión estamos haciendo el heurístico más informado. Manteniendo esta información sabremos si hemos de volver a regenerar esa rama olvidada, si pasa a ser la de mejor coste.

El efecto que tiene esta exploración es ir refinando la estimación que se tiene de la función heurística en los nodos que se mantienen abiertos, de manera que el número de reexpansiones se irá reduciendo paulatinamente ya que el coste guardado será cada vez más cercano al coste real. Se puede ver en algoritmo 3.4.

Ejemplo 3.4 En la figura 3.4 se puede ver como evoluciona la búsqueda en un ejemplo sencillo, en cada nodo aparece el valor de la función f :

Podemos ver que cuando llega a la tercera iteración el nodo con mejor coste es el de 12 y por lo tanto la recursividad vuelve hasta ese nodo borrando la rama explorada, pero actualizando cada padre con el mejor coste encontrado, en este caso 17. En la cuarta iteración se encuentra que ahora es el nodo con coste 17 el de mejor coste, por lo que el camino vuelve a regenerarse.

Por lo general, el número de reexpansiones de este algoritmo es menor que el de IDA*, pero puede depender de las características del espacio de búsqueda como por ejemplo la longitud de los ciclos que pueda haber. Este algoritmo también impone un coste lineal al espacio, por lo que también se pueden generar bastantes reexpansiones.

Otra alternativa diferente viene de una modificación del A*, es el algoritmo llamado **A* con memoria limitada (memory bound A*)**. Este algoritmo utiliza la estrategia de exploración de A* pero siguiendo una estrategia de memorización de caminos parecida al **best first recursivo**. Éste almacena todos los caminos que le caben en el tamaño de memoria que se impone y no solo el actual, de manera que se ahorra regenerar ciertos caminos. Cuanta más memoria permitamos, menos regeneraciones de caminos haremos ya que nos los encontraremos almacenados.

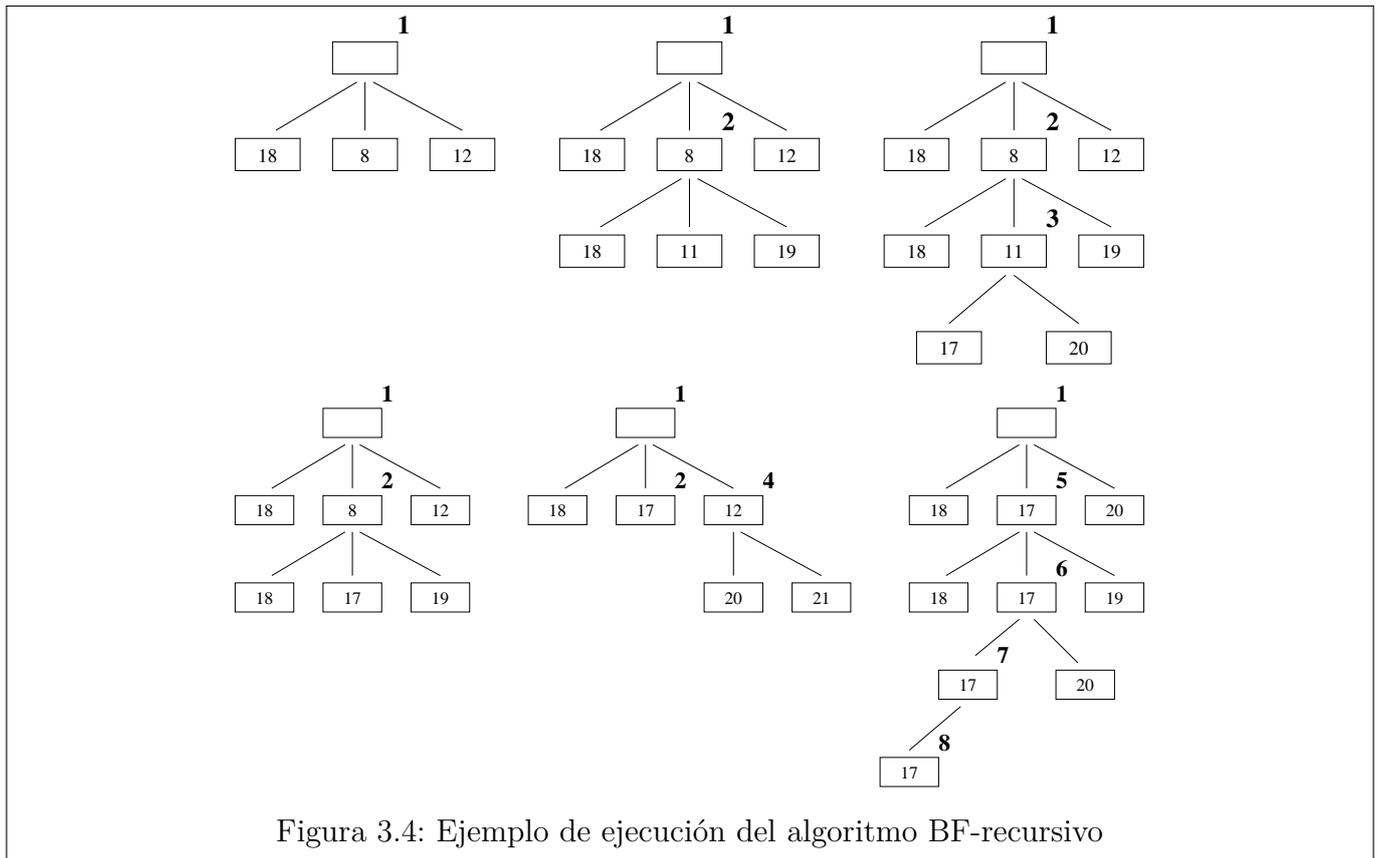


Figura 3.4: Ejemplo de ejecución del algoritmo BF-recursivo

El algoritmo elimina los caminos peores en el momento en el que ya no caben más, guardando en los nodos la información suficiente que le permita saber cuando se han de regenerar. El aporte extra de memoria permite reducir bastante las reexpansiones, con el consiguiente ahorro de tiempo.

El algoritmo es capaz de hallar una solución, si el camino completo cabe en la cantidad de memoria que hemos impuesto. Esto quiere decir que, si tenemos una idea aproximada de la longitud de la solución podemos ajustar a priori la cantidad de memoria que permitiremos usar al algoritmo.

Este algoritmo es mucho mas complejo que los anteriores y existen diferentes versiones. Principalmente la complejidad viene de como se han de reorganizar los caminos memorizados cuando se han de borrar la memoria y la propagación de los costes a los nodos ascendientes que se mantienen.

4.1 El tamaño importa, a veces

Los algoritmos que hemos visto en el capítulo anterior nos sirven siempre que podamos plantear el problema como la búsqueda de un camino en un espacio de estados. Pero nos podemos encontrar con problemas en los que:

- Este planteamiento es demasiado artificial, ya que no tenemos realmente operadores de cambio de estado, o no hay realmente 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 del espacio 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 guiado por la función heurística. Si no es así, deberemos usar otros algoritmos más exhaustivos que nos aseguren hallar una solución.

En este planteamiento lo que suponemos es que podemos navegar dentro del espacio de posibles soluciones realizando operaciones sobre ellas que nos pueden ayudar a mejorarlas. El coste de estas operaciones no será relevante ya que lo que nos importa es la calidad de la solución¹, en este caso lo que nos importará es esa la calidad. Deberemos disponer de una función que según algún criterio (dependiente del dominio) nos permita ordenarlas.

El planteamiento de los problemas es parecido al que consideramos al hablar del espacio de estados, tenemos los siguientes elementos:

- Un estado inicial, que en este caso será una solución completa. Esto nos planteará el problema adicional de como hallarla con un coste relativamente bajo y el evaluar desde donde empezaremos a buscar, si desde una solución relativamente buena, desde la peor, desde una al azar, ...
- Unos operadores de cambio de la solución, que en este caso manipularán soluciones completas y que no tendrán un coste asociado.

¹De hecho muchas veces estas operaciones no se refieren a operaciones reales en el problema, sino a formas de manipular la solución.

- Una función de calidad, que nos medirá lo buena que es una solución y nos permitirá guiar la búsqueda, pero teniendo en cuenta que este valor no nos indica cuanto nos falta para encontrar la solución que buscamos. Ya que no representa un coste, es una manera de comparar la calidad entre las soluciones vecinas e indicarnos cual es la dirección que lleva a soluciones mejores.

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 tendremos un estado final definido.

Para buscar una analogía con un tema conocido, se puede asimilar la búsqueda local con la búsqueda de óptimos en funciones. En este caso la función a optimizar² será la función de calidad de la solución, la función se definirá en el espacio de soluciones generado por la conectividad que inducen los operadores entre ellas.

Desafortunadamente, las funciones que aparecerán no tendrán una expresión analítica global, ni tendrán las propiedades que nos permitirían aplicar los algoritmos de búsqueda de óptimos en funciones. Estas funciones pueden ser de cualquier tipo, y no están limitadas de la manera que lo están las funciones usadas por los algoritmos de búsqueda heurística. Sus valores pueden ser tanto positivos, como negativos y, obviamente, su valor en la solución no tiene por que ser cero.

El planteamiento será pues bastante diferente. Ya que no tenemos una expresión analítica global, deberemos explorar la función de calidad utilizando solamente lo que podamos obtener de los vecinos de una solución, ésta deberá permitir decidir por donde seguir buscando el óptimo. El nombre de búsqueda local viene precisamente de que solo utilizamos información local durante el proceso de hallar la mejor solución.

4.2 Tu sí, vosotros no

Enfrentados a problemas con tamaños de espacio de búsqueda inabordables de manera exhaustiva, nos hemos de plantear estrategias diferentes a las utilizadas hasta ahora.

En primer lugar, tendremos pocas posibilidades de guardar información para recuperar caminos alternativos dado el gran número de alternativas que habrá. Esto nos obligará a imponer unas restricciones de espacio limitadas, lo que nos llevará a tener que decidir que nodos debemos explorar y cuales descartar sin volver a considerarlos.

Esto nos lleva a la familia de algoritmos conocida como de **ramificación y poda (branch and bound)**. Esta familia de algoritmos limita el número de elementos que guardamos como pendientes de explotar olvidando los que no parecen prometedores. Estos algoritmos permiten mantener una memoria limitada, ya que desprecian parte del espacio de búsqueda, pero se arriesgan a no hallar la mejor solución, ya que esta puede estar en el espacio de búsqueda que no se explora.

De entre los algoritmos de *ramificación y poda*, los más utilizados son los denominados de **ascenso de colinas (Hill-climbing)**. Existen diferentes variantes que tienen sus ventajas e inconvenientes.

Por ejemplo, el denominado **ascenso de colinas simple**, que consiste en elegir siempre el primer operador que suponga una mejora respecto al nodo actual, de manera que no exploramos todas las posibilidades accesibles, ahorrándonos el explorar cierto número de descendientes. La ventaja es que es más rápido que explorar todas las posibilidades, la desventaja es que hay más probabilidad de no alcanzar las soluciones mejores.

El más utilizado es el **ascenso de colinas por máxima pendiente (steepest ascent hill climbing)**. Esta variante expande todos los posibles descendientes de un nodo y elige el que suponga la máxima mejora respecto al nodo actual. Con esta estrategia suponemos que la mejor solución la

²Hablamos de optimizar, y no de minimizar o maximizar, ya que la diferencia entre ellas es simplemente un cambio de signo en la función a optimizar.

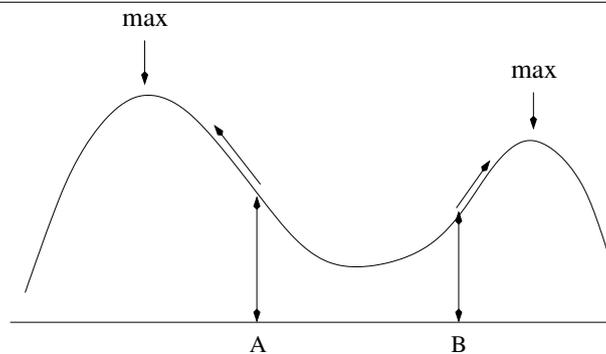
Algoritmo 4.1 Algoritmo Steepest Ascent Hill Climbing**Algoritmo:** Hill ClimbingActual \leftarrow Estado_inicialfin \leftarrow falso**mientras** no fin **hacer** Hijos \leftarrow generar_sucesores(Actual) Hijos \leftarrow ordenar_y_eliminar_peores(Hijos, Actual) **si** no vacio?(Hijos) **entonces** Actual \leftarrow Escoger_mejor(Hijos) **sino** fin \leftarrow cierto **fin****fin**

Figura 4.1: El óptimo depende del punto de inicio

encontraremos a través del sucesor que mayor diferencia tenga respecto a la solución actual, siguiendo una política avariciosa. Como veremos más adelante esta estrategia puede ser arriesgada.

El algoritmo que implementa este último es el algoritmo 4.1.

En este algoritmo la utilización de memoria es nula, ya que solo tenemos en cuenta el nodo mejor³. Se pueden hacer versiones que guarden caminos alternativos que permitan una vuelta atrás en el caso de que consideremos que la solución a la que hemos llegado no es suficientemente buena, pero hemos de imponer ciertas restricciones de memoria, si no queremos tener un coste espacial demasiado grande. Comentaremos estos algoritmos más adelante.

La estrategia de este algoritmo hace que los problemas que tiene, vengan principalmente derivados por las características de las funciones heurísticas que se utilizan.

Por un lado, hemos de considerar que el algoritmo parará la exploración en el momento en el que no se encuentra ningún nodo accesible mejor que el actual. Dado que no mantenemos memoria del camino recorrido nos será imposible reconsiderar nuestras decisiones, de modo que si nos hemos equivocado, la solución a la que llegaremos será la óptima.

Esta circunstancia es probable, ya que seguramente la función heurística que utilicemos tendrá óptimos locales y, dependiendo de por donde empecemos a buscar, acabaremos encontrando uno u otro, como se puede ver en la figura 4.1. Si iniciamos la búsqueda desde el punto A o el B, el máximo que alcanzaremos será diferente.

Esta circunstancia se puede mejorar mediante la ejecución del algoritmo cierto número de veces desde distintos puntos escogidos aleatoriamente y quedándonos con la mejor exploración. A esta

³Hay que recordar que en este tipo de búsqueda el camino no nos importa, por lo que el gasto en espacio es constante.

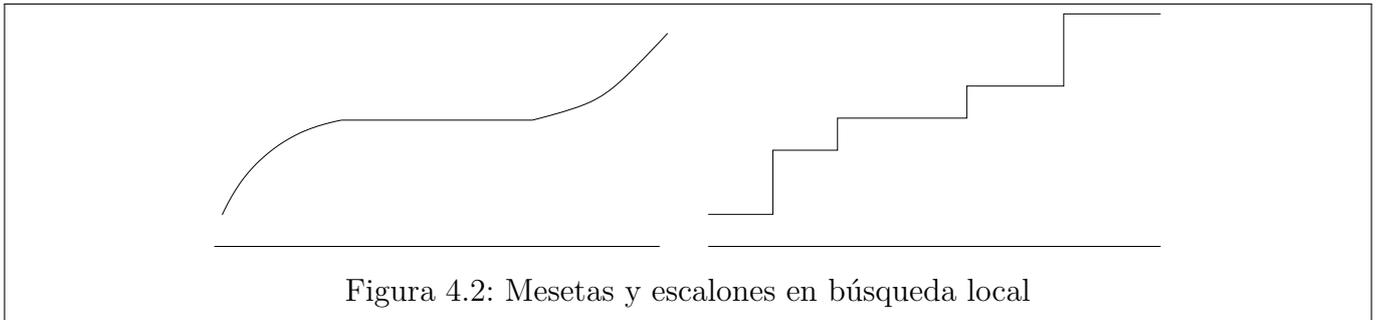


Figura 4.2: Mesetas y escalones en búsqueda local

estrategia se la denomina búsqueda por ascenso con **reinicio aleatorio** (**random restarting hill climbing**). Muchas veces esta estrategia es más efectiva que métodos más complejos que veremos a continuación y resulta bastante barata. La única complicación está en poder generar los distintos puntos iniciales de búsqueda, ya que en ocasiones el problema no permite hallar soluciones iniciales con facilidad.

Otro problema con el que se encuentran estos algoritmos son zonas del espacio de búsqueda en las que la función heurística no es informativa. Un ejemplo son las denominadas mesetas y las funciones en escalón (figura 4.2), en las que los valores de los nodos vecinos al actual tienen valores iguales, y por lo tanto una elección local no nos permite decidir el camino a seguir.

Evitar estos problemas requiere extender la búsqueda a más allá de los vecinos inmediatos para obtener la información suficiente para encaminar la búsqueda. Desgraciadamente esto supone un coste adicional en cada iteración.

Una alternativa es permitir que el algoritmo guarde parte de los nodos visitados para proseguir la búsqueda por ellos en el caso de que el algoritmo se quede atascado en un óptimo local. El algoritmo más utilizado es el denominado **búsqueda en haces** (**beam search**). En este algoritmo se van guardando un número N de las mejores soluciones, expandiendo siempre la mejor de ellas. De esta manera no se sigue un solo camino sino N .

Las variantes del algoritmo están en cómo se escogen esas N soluciones que se mantienen. Si siempre se substituyen las peores soluciones de las N por los mejores sucesores del nodo actual, caemos en el peligro de que todas acaben estando en el mismo camino, reduciendo la capacidad de volver hacia atrás, así que el poder mantener la variedad de las soluciones guardadas es importante.

Está claro que cuantas más soluciones guardemos más posibilidad tendremos de encontrar una buena solución, pero tendremos un coste adicional tanto en memoria como en tiempo, ya que tendremos que calcular con que sucesores nos quedamos y que soluciones guardadas substituímos. Su implementación se puede ver en el algoritmo 4.2.

El algoritmo acaba cuando ninguno de los sucesores mejora a las soluciones guardadas, esto quiere decir que todas las soluciones son un óptimo local. Como veremos más adelante esta técnica de búsqueda tiene puntos en común con los algoritmos genéticos.

4.3 Un mundo de posibilidades

La estrategia de búsqueda del algoritmo hill climbing y sus variantes es la más simple que se puede utilizar. El mayor problema es que es demasiado optimista y asume que hay un camino más o menos directo (siempre ascendente) a los óptimos locales. En problemas difíciles esto no tiene por que ser así, y hacen falta estrategias un poco más exhaustiva, explorando más nodos que solo el camino directo y que sepan aprovechar la información que vamos encontrando por el camino.

Varias son las estrategias que podemos añadir a la estrategia básica del hill climbing que pueden mejorar sustancialmente sus resultados. La primera es explorar no solo los vecinos mejores, sino también otros que pueden ser peores, pero que pueden ocultar un camino que es mejor que el que

Algoritmo 4.2 Algoritmo Beam Search

```

Algoritmo: Beam Search
Actual ← Estado_inicial
Soluciones_actuales.añadir(Estado_inicial)
fin ← falso
mientras no fin hacer
    Hijos ← generar_sucesores(Actual)
    soluciones_actuales.actualizar_mejores(Hijos)
    si soluciones_actuales.algun_cambio?() entonces
        | Actual ← soluciones_actuales.escoger_mejor()
    sino
        | fin ← cierto
    fin
fin

```

podemos encontrar si seguimos directamente la pendiente ascendente de la función heurística. Para decidir qué nodos peores visitar necesitamos una estrategia de decisión, una posibilidad es la que plantea el algoritmo *simulated annealing* que veremos en la siguiente sección.

Otra posibilidad es usar información de más alto nivel que podamos ir encontrando durante la búsqueda, como por ejemplo, qué operaciones son las que consiguen mejorar más las soluciones (o no lo consiguen) de manera que podamos concentrarnos más en algunas operaciones que en otras (reduciendo así el espacio de búsqueda, o que partes de la descripción de la solución aportan más a la calidad de la solución o qué partes nunca la mejoran, de manera que nos podamos concentrar en operaciones que se centren en esos elementos de la solución. Estas y otras estrategias parecidas son las que utiliza el algoritmo *tabú search*, este algoritmo puede prohibir o permitir operaciones sobre la solución en función de la información que se vaya obteniendo durante la búsqueda.

Una última posibilidad que ha dado lugar a muchos algoritmos es el uso de una población de soluciones en lugar de una sola solución, siguiendo la idea del algoritmo *beam search*. El tener una población de soluciones nos permite ver más parte del espacio de búsqueda, siendo más exhaustivos y, además, obtener más información que nos permitirá concentrarnos más en soluciones con ciertas características o el evitar parte del espacio de búsqueda que no tenga soluciones buenas.

Muchos de los métodos que siguen esta estrategia están inspirados en analogías biológicas o físicas. Los más conocidos son los algoritmos evolutivos, que incluyen a los algoritmos genéticos que veremos en la última sección, pero que también incluyen a otros algoritmos que se basan en conceptos inspirados en la evolución de las especies y ecología.

Otra familia de algoritmos que usan también una estrategia de población es la denominada *swarm intelligence*, incluye algoritmos que están inspirados en biología como el algoritmo *Particle Swarm Optimization*, que se inspira en el comportamiento de las bandadas de pájaros y los bancos de peces, el *Ant Colony Optimization* que se inspira en la forma que usan las hormigas de solucionar problemas (son buenos para la resolución de problemas que se pueden representar como caminos en grafos), además de algoritmos inspirados en el comportamiento de abejas, luciérnagas, cucos, monos, el sistema inmunitario... También hay algoritmos inspirados en fenómenos físicos como *Intelligent Water Drops* que simula la interacción de las gotas de agua, *River formation dynamics*, que usa la estrategia de formación de los ríos, *Gravitational search algorithm*, que usa la interacción de la fuerza de gravedad entre partículas, ...

La aplicación de todos estos algoritmos se ha ido extendiendo más allá de los problemas de inteligencia artificial y se aplican a cualquier problema que involucre la optimización de una función y que no se pueda resolver por métodos clásicos. De hecho, todos estos algoritmos se incluyen en un

área de computación conocida como *metaheurísticas*

4.4 Demasiado calor, demasiado frío

Se han planteado algoritmos alternativos de búsqueda local que se han mostrado efectivos en algunos dominios en los que los algoritmos de búsqueda por ascenso se quedan atascados enseguida en óptimos no demasiado buenos.

El primero que veremos es el denominado **templado simulado** (**simulated annealing**). Este algoritmo está inspirado en un fenómeno físico que se observa en el templado de metales y en la cristalización de disoluciones.

Todo conjunto de átomos o moléculas tiene un estado de energía que depende de cierta función de la temperatura del sistema. A medida que lo vamos enfriando, el sistema va perdiendo energía hasta que se estabiliza. El fenómeno físico que se observa es que dependiendo de como se realiza el enfriamiento, el estado de energía final es muy diferente.

Por ejemplo, si una barra de metal se enfría demasiado rápido la estructura cristalina a la que se llega al final está poco cohesionada (un número bajo de enlaces entre átomos) y ésta se rompe con facilidad. Si el enfriamiento se realiza más lentamente, el resultado final es totalmente diferente, el número de enlaces entre los átomos es mayor y se obtiene una barra mucho más difícil de romper.

Durante este enfriamiento controlado, se observa que la energía del conjunto de átomos no disminuye de manera constante, sino que a veces la energía total puede ser mayor que en un momento inmediatamente anterior. Esta circunstancia hace que el estado de energía final que se alcanza sea mejor que cuando el enfriamiento se hace rápidamente. A partir de este fenómeno físico, podemos derivar un algoritmo que permitirá en ciertos problemas hallar soluciones mejores que los algoritmos de búsqueda por ascenso.

En este algoritmo el nodo siguiente a explorar no será siempre el mejor descendiente, sino que lo elegiremos aleatoriamente en función de los valores de unos parámetros entre todos los descendientes (los buenos y los malos). Una ventaja de este algoritmo es que no tenemos que generar todos los sucesores de un nodo, basta elegir un sucesor al azar y decidir si continuamos por él o no. El algoritmo de templado simulado se puede ver como una versión estocástica del algoritmo de búsqueda por ascenso.

El algoritmo de templado simulado intenta transportar la analogía física al problema que queremos solucionar. En primer lugar denominaremos función de **energía** a la función heurística que nos mide la calidad de una solución. Tendremos un parámetro de control que denominaremos **temperatura**, que nos permitirá controlar el funcionamiento del algoritmo.

Tendremos una función que dependerá de la temperatura y de la diferencia de calidad entre el nodo actual y un sucesor. Ésta gobernará la elección de los sucesores, su comportamiento será el siguiente:

- Cuanta mayor sea la temperatura, más probabilidad habrá de que al generar como sucesor un estado peor éste sea elegido
- La elección de un estado peor estará en función de su diferencia de calidad con el estado actual, cuanto más diferencia, menos probabilidad de elegirlo.

El último elemento es la **estrategia de enfriamiento**. 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 desde 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. Habrá que decidir

Algoritmo 4.3 Algoritmo Simulated Annealing**Algoritmo:** Simulated Annealing

Partimos de una temperatura inicial

mientras la temperatura no sea cero **hacer**

// Paseo aleatorio por el espacio de soluciones

para un numero prefijado de iteraciones **hacer** Enuevo \leftarrow Genera_sucesor_al_azar(Eactual) $\Delta E \leftarrow f(Eactual) - f(Enuevo)$ **si** $\Delta E > 0$ **entonces** | Eactual \leftarrow Enuevo **sino** | con probabilidad $e^{\Delta E/T}$: Eactual \leftarrow Enuevo **fin** **fin**

Disminuimos la temperatura

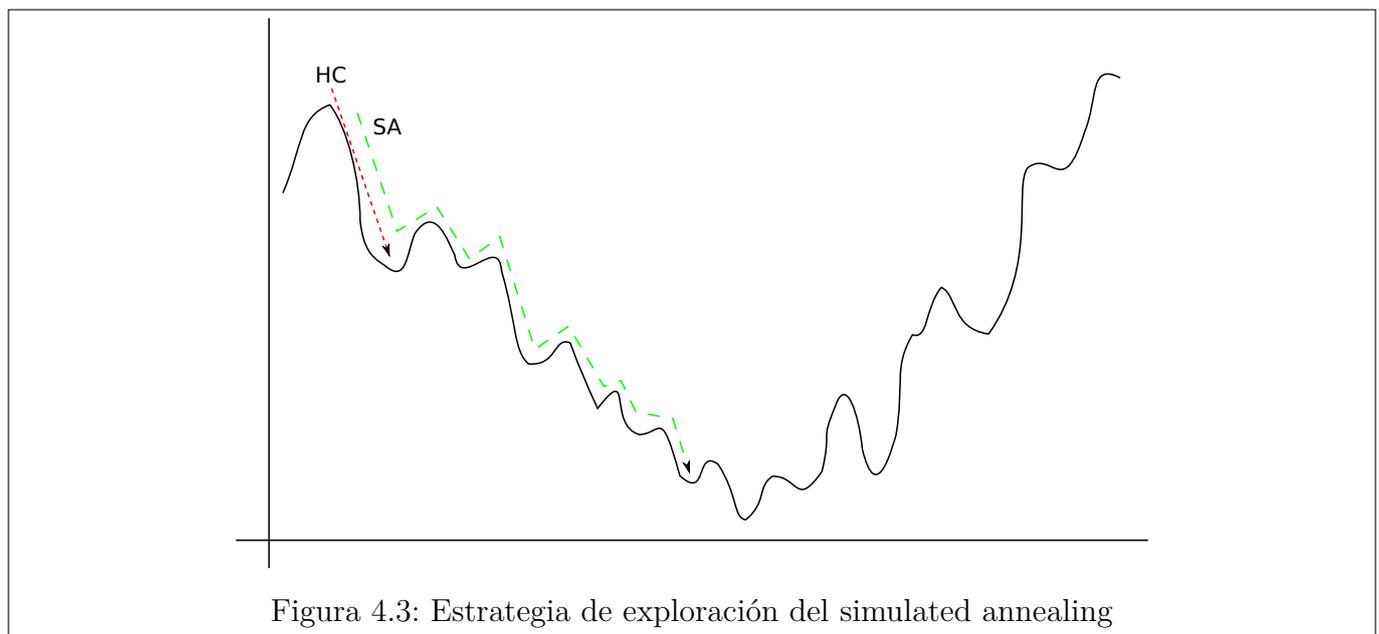
fin

Figura 4.3: Estrategia de exploración del simulated annealing

experimentalmente cual es la temperatura inicial más adecuada y forma más adecuada de hacer que vaya disminuyendo.

Como en la analogía física, 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.

El algoritmo que implementa esta estrategia es el algoritmo 4.3. En la figura 4.3 podemos ver el comportamiento que tendría el simulated annealing comparado con el de hill climbing.

Este algoritmo se adapta muy bien a problemas de optimización combinatoria (configuración óptima de elementos) y continua (punto óptimo en un espacio N-dimensional). Está indicado para problemas con un espacio de búsqueda grande en los que el óptimo está rodeado de muchos óptimos locales⁴, ya que a este algoritmo le será más fácil escapar de ellos.

Se puede utilizar también para problemas en los que encontrar una heurística discriminante es difícil (una elección aleatoria es tan buena como otra cualquiera), ya que el algoritmo de ascenso

⁴Un área en la que estos algoritmos han sido muy utilizados es la del diseño de circuitos VLSI.

de colinas no tendrá mucha información con la que trabajar y se quedará atascado enseguida. En cambio el templado simulado podrá explorar más espacio de soluciones y tendrá mayor probabilidad de encontrar una solución buena.

El mayor problema de este algoritmo será determinar los valores de los parámetros, y requerirá una importante labor de experimentación que dependerá de cada problema. Estos parámetros variarán con el dominio del problema e incluso con el tamaño de la instancia del problema.

4.5 Cada oveja con su pareja

Otra analogía que ha dado lugar un conjunto de algoritmos de búsqueda local bastante efectivos es la selección natural como mecanismo de adaptación de los seres vivos. Esta analogía se fija en que los seres vivos se adaptan al entorno gracias a las características heredadas de sus progenitores, en que las posibilidades de supervivencia y reproducción son proporcionales a la bondad de esas características y en que la combinación de buenos individuos puede dar lugar a individuos mejor adaptados.

Podemos trasladar estos elementos a la búsqueda local identificando las soluciones con individuos y donde una función de calidad indicará la bondad de la solución, es decir, su adaptación a las características del problema. Dispondremos de unos operadores de búsqueda que combinarán buenas soluciones con el objetivo de obtener soluciones mejores como resultado.

El ejemplo que veremos de algoritmos que utilizan esta analogía es el de los **algoritmos genéticos**⁵. Se trata de una familia bastante amplia de algoritmos, nosotros solo veremos el esquema básico.

Estos algoritmos son bastante más complejos que los algoritmos que hemos visto hasta ahora y requieren más elementos y parámetros, por lo que su utilización requiere cierta experiencia y mucha experimentación. Los requisitos básicos para usarlos son:

- Dar una codificación a las características de las soluciones. Las soluciones se representan de una manera especial para que después se puedan combinar. Por lo general se utilizan cadenas binarias que codifican los elementos de la solución, por analogía a esta codificación se la denomina **gen**⁶.
- Tener una función que mida la calidad de la solución. Se tratará de la función heurística que se utiliza en todos los algoritmos que hemos visto. En el área de algoritmos genéticos a esta función se la denomina **función de adaptación (fitness)**.
- Disponer de operadores que combinen las soluciones para obtener nuevas soluciones. Los operadores que utilizan los algoritmos genéticos son bastante fijos y están *inspirados* en las formas de reproducción celular.
- Decidir el número de individuos inicial. Un algoritmo genético no trata un solo individuo, sino una población, se ha de decidir cuan numerosa ha de ser.
- Decidir una estrategia para hacer la combinación de individuos. Siguiendo la analogía de la selección natural, solo se reproducen los individuos más aptos, hemos de decidir un criterio para emparejar a los individuos de la población en función de su calidad.

⁵Existen también los denominados algoritmos evolutivos, son menos conocidos, pero suelen ser bastante efectivos en ciertas aplicaciones.

⁶Esta no tiene por que ser siempre la representación más adecuada y de hecho a veces una representación binaria hace que el problema no se pueda solucionar eficientemente.

Vamos a detallar un poco más cada uno de estos elementos.

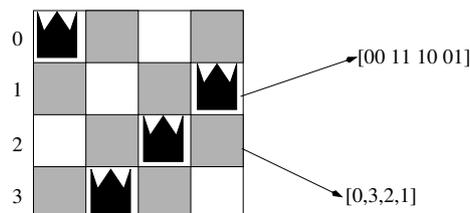


El uso de los algoritmos genéticos presenta ciertas ventajas prácticas respecto a los algoritmos anteriores. Estamos redefiniendo como se representan los problemas, todos tendrán una representación común independientemente de su naturaleza, por lo que solo nos deberemos de preocupar de como codificar 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.

4.5.1 Codificación

Como hemos mencionado, la forma más habitual de codificar los individuos para utilizar un algoritmo genético es transformarlos a una cadena de bits. Cada bit o grupo de bits codificará una característica de la solución. Evidentemente, no existe un criterio preestablecido sobre como realizarla.

La ventaja de esta codificación es que los operadores de modificación de la solución son muy sencillos de implementar. En la siguiente figura se puede ver un ejemplo de codificación para el problema de las N reinas siendo N igual a 4:



En la codificación binaria se ha asignado una pareja de bits a cada una de las reinas, representando la fila en la que están colocadas y se han concatenado. Alternativamente se podría utilizar una representación no binaria utilizando simplemente el número de fila y haciendo una lista con los valores.

Como se ha comentado, no siempre la codificación binaria es la más adecuada, ya que las características del problema pueden hacer que el cambio de representación haga más complicado el problema.

Una ventaja de la codificación binaria es que 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.



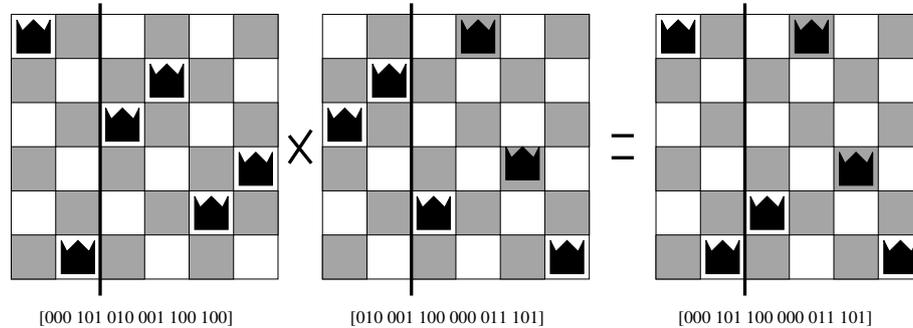
A veces es importante pensar bien como codificamos los estados, hay que tener en cuenta que la codificación binaria impone la conectividad entre los estados y que, al hacer la transformación, estados que antes eran vecinos ahora no lo serán. Puede ser interesante hacer que la codificación mantenga esa vecindad para obligar a que el algoritmo haga la exploración del espacio de búsqueda de cierta manera.

También a veces nos interesa que diferentes partes de la codificaciones correspondan a partes específicas del estado y así obtener patrones que puedan mantenerse y propagarse de manera más sencilla.

4.5.2 Operadores

Los operadores que utilizan estos algoritmos para la exploración del espacio de búsqueda se pueden agrupar en dos tipos básicos. Los operadores de **cruce** (**crossover**) y los de **mutación** (**mutation**).

Los operadores de cruce se aplican a una pareja de individuos, su efecto consiste en intercambiar parte de la información de la codificación entre los individuos. Existe una gran variedad de formas en las que se puede realizar este intercambio. La más sencilla es la denominada **cruce por un punto (one point crossover)**. Se aplica solo a codificaciones binarias, este operador consiste en elegir un punto al azar en la codificación e intercambiar los bits de los dos individuos a partir de ese punto. En la siguiente figura hay un ejemplo de cruce por un punto en el problema de las N reinas:



Otra variante muy utilizada es el **cruce por dos puntos (two points cross over)**, en el que el intercambio se realiza eligiendo dos puntos en la codificación e intercambiando los bits entre esos dos puntos.

Aparte de estos dos operadores existen muchos otros que tienen efectos específicos y que funcionan mejor en ciertas circunstancias (uniform crossover, random crossover, ...). Si la codificación no es binaria, los operadores se han de diseñar de manera específica para la representación escogida. También hay que pensar a veces en la eficiencia de la aplicación de los operadores, cuando tenemos que procesar muchas soluciones nos puede interesar operadores que puedan traducirse a operaciones aritméticas simples entre tiras de bits (or, and, xor, shifts, ...).

Los operadores de mutación se aplican a un solo individuo y consisten en cambiar parte de la codificación de manera aleatoria. El más sencillo es elegir un bit al azar y cambiar su signo.

Estos operadores tienen una probabilidad de aplicación asociada que será un parámetro del algoritmo. Tendremos una probabilidad de cruce que indicará cuando una pareja elegida de individuos intercambia su información y una probabilidad de mutación. 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.



La probabilidad de aplicación de los operadores puede ser algo muy delicado, ya que cambia la forma en la que se hace la exploración al cambiar las características de la población. 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 mas lenta y puede tardar mucho en converger.

Con al mutación pasa algo parecido. Esta es esencial para que el algortimo 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.

4.5.3 Combinación de individuos

Estos algoritmos no tratan las soluciones de manera individual, sino en grupo. Denominaremos **población** al conjunto de soluciones que tenemos en un momento específico. El efecto de mantener un conjunto de soluciones a la vez es hacer una exploración en paralelo del espacio de búsqueda desde diferentes puntos.

Un parámetro más que deberemos decidir es el tamaño de la población de soluciones. Este es crítico respecto a la capacidad de exploración. Si el número de individuos es demasiado grande, el coste por iteración puede ser prohibitivo, pero si el tamaño es demasiado pequeño, es posible que no lleguemos a una solución demasiado buena, al no poder ver suficiente espacio de búsqueda.

Cada iteración de un algoritmo genético es denominada una **generación**, a cada generación los individuos se emparejan y dan lugar a la siguiente generación de individuos. Es clave la manera en la que decidimos como se emparejan los individuos.

La forma habitual de pasar de una generación a otra, es crear lo que se denomina una **generación intermedia** que estará compuesta por las parejas que se van a combinar. El número de individuos de esta población es igual al del tamaño original de la población. La elección de los individuos que forman parte de esta generación intermedia se puede hacer de muchas maneras, por ejemplo:

- Escoger un individuo de manera proporcional a su valor de evaluación de la función de fitness, de manera que los individuos mejores tendrán más probabilidad de ser elegidos. Esto puede tener el peligro de que unos pocos individuos pueden ser elegidos muchas veces.
- Se establecen N torneos entre parejas de individuos escogidas al azar, el ganador de cada emparejamiento es el individuo con mejor valor en la función de fitness. Esta opción equilibra mejor la aparición de los mejores individuos.
- Se define un ranking lineal entre individuos según su función de calidad, estableciendo un máximo de posibles apariciones de un individuo, de esta manera que la probabilidad de aparición de los individuos no esté sesgada por los individuos mejores.

Con estos mecanismos de elección habrá individuos que aparezcan más de una vez e individuos que no aparezcan⁷. Cada mecanismo de selección de individuos tiene sus ventajas e inconvenientes. El primero es el más simple, pero 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. Se ha visto experimentalmente que el asegurar la variedad en la población permite encontrar mejores soluciones⁸.

4.5.4 El algoritmo genético canónico

Existe una gran variedad de algoritmos genéticos, pero todos parten de un funcionamiento básico que llamaremos el **algoritmo genético canónico**, los pasos que realiza este algoritmo básico son estos:

1. Se parte de una población de N individuos generada aleatoriamente
2. Se escogen N individuos de la generación actual para la generación intermedia (según el criterio escogido)
3. Se emparejan los individuos y para cada pareja
 - a) Con una probabilidad P_{cruce} se aplica el operador de cruce a cada pareja de individuos y se obtienen dos nuevos individuos que pasarán a la nueva generación. Con una probabilidad $1 - P_{cruce}$ se mantienen la pareja original en la siguiente generación
 - b) Con una probabilidad $P_{mutacion}$ se mutan los individuos cruzados

⁷Esto refleja lo que sucede en la selección natural, los más aptos tienen una mayor probabilidad de tener descendencia y los menos aptos pueden no llegar a reproducirse.

⁸También es algo que se observa en la naturaleza, las poblaciones aisladas o las especies muy especializadas tienen un pobre acervo genético y no pueden adaptarse a cambios en su entorno.

4. Se substituye la población actual con los nuevos individuos, que forman la nueva generación
5. Se itera desde el segundo paso hasta que la población converge (la función de fitness de la población no mejora) o pasa un número específico de generaciones

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. Si ésta es muy alta, cada generación será muy diferente, por lo que puede degenerar en una búsqueda aleatoria.

La mutación es un mecanismo para forzar la variedad en la población, pero una probabilidad de mutación demasiado alta puede dificultar la convergencia.

4.5.5 Cuando usarlos

Los algoritmos genéticos han demostrado su eficacia en múltiples aplicaciones, pero es difícil decir si serán efectivos o no en una aplicación concreta. Por lo general suelen funcionar mejor que los algoritmos de búsqueda por ascenso cuando no se tiene una buena función heurística para guiar el proceso. Pero si se dispone de una buena heurística, estos algoritmos no nos obtendrán mejores resultados y el coste de configurarlos adecuadamente no hará rentable su uso.

Su mayor inconveniente es la gran cantidad de elementos que hemos de ajustar para poder aplicarlos, la codificación del problema, el tamaño de la población, la elección de los operadores, las probabilidades de cruce y mutación, la forma de construir la siguiente generación, ... Siempre nos encontraremos con la duda de si el algoritmo no funciona porque el problema no se adapta al funcionamiento de estos algoritmos, o es que no hemos encontrado los parámetros adecuados para hacer que funcionen bien.

5.1 Tú contra mi o yo contra ti

Hasta ahora habíamos supuesto problemas en los que un solo *agente* intenta obtener un objetivo, en nuestro caso la resolución de un problema. Pero existen casos en los que diferentes agentes compiten por un objetivo que solamente uno de ellos puede alcanzar.

Este tipo de problemas los englobaremos dentro de la categoría de juegos¹. No veremos algoritmos para cualquier tipo de juego, sino que nos restringiremos a los que cumplen un conjunto de propiedades:

- Son bipersonales, es decir, solo hay dos agentes involucrados, y estos juegan alternativamente.
- Todos los participantes tienen conocimiento perfecto, no hay ocultación de información por parte de ninguno de ellos.
- El juego es determinista, no interviene el azar en ninguno de sus elementos.

Los elementos que intervienen en este tipo de problemas se pueden representar de manera parecida a la búsqueda en espacio de estados, tendremos:

- Un estado, que indica las características del juego en un instante específico
- Un estado inicial, que determinará desde donde se empieza a jugar y quién inicia el juego
- Un estado final, o unas características que debe cumplir, que determinarán quién es el ganador
- Unos operadores de transformación de estado que determinarán las jugadas que puede hacer un agente desde el estado actual. Supondremos que los dos jugadores disponen de los mismos operadores

Identificaremos a cada jugador como el jugador **MAX** y el jugador **MIN**. MAX será el jugador que inicia el juego, nos marcaremos como objetivo el encontrar el conjunto de movimientos que ha de hacer el jugador MAX para que llegue al objetivo (ganar) independientemente de lo que haga el jugador MIN².

Se puede observar que este escenario es bastante diferente del que se plantean los algoritmos de los capítulos anteriores. Nos veremos obligados a revisar el concepto de solución y de óptimo. De hecho estamos pasando de un escenario en el que poseemos todo el conocimiento necesario para resolver un problema desde el principio hasta el fin sin tener que observar la reacción del entorno, a un escenario donde nuestras decisiones se ven influidas por el entorno. En los algoritmos que veremos supondremos que podemos prever hasta cierto punto como reaccionará ese entorno, pero evidentemente no siempre tendrá por que ser ese el caso.

¹La competición no solamente aparece en el caso de los juegos, pero los algoritmos que veremos también son aplicables.

²No es que nos caiga mal el jugador MIN, lo que pasa es que los algoritmos serán los mismos para los dos, ya que desde la perspectiva de MIN, él es MAX y su oponente es MIN.

5.2 Una aproximación trivial

Para poder obtener el camino que permite a MAX llegar a un estado ganador, no tenemos más que explorar todas las posibles jugadas que puede hacer cada jugador, hasta encontrar un camino que lleve a un estado final. Para poder obtenerlo podemos utilizar un algoritmo de búsqueda en profundidad que explore exhaustivamente todas las posibilidades. Usamos un algoritmo de búsqueda en profundidad ya que sus restricciones de espacio son lineales.

El algoritmo explorará hasta llegar a una solución evaluándola a $+1$ si es un estado en el que gana MAX, o a -1 si es un estado en el que gana MIN, este valor se propagará hasta el nivel superior. A medida que se acaben de explorar los descendientes de un nivel el valor resultante de su exploración se seguirá propagando a los niveles superiores.

Cada nivel elegirá el valor que propaga a su ascendiente dependiendo de si corresponde a un nivel del jugador MAX o del jugador MIN. En los niveles de MAX se elegirá el valor mayor de todos los descendientes, en los niveles de MIN se elegirá el mínimo. En el momento en el que se propague un $+1$ a la raíz significará que hemos encontrado el camino que permite ganar a MAX independientemente de las jugadas de MIN. En este momento podemos parar la exploración ya que hemos cubierto su objetivo, encontrar una secuencia de pasos ganadora.

En la figura 5.1 podemos ver el espacio de búsqueda que generaría un juego hipotético (los cuadrados corresponden a niveles de MAX y los círculos a niveles de MIN), se han marcado con $+1$ las jugadas en las que gana el jugador MAX y con -1 las jugadas en la que gana el jugador MIN

Si hacemos el recorrido en profundidad propagando los valores de las jugadas terminales, tendríamos el resultado que se muestra en la figura 5.2. En ella podemos ver marcado el camino que permite ganar a MAX.

Desgraciadamente este algoritmo solo lo podemos aplicar a juegos triviales, ya que el tiempo que necesitaríamos para hacer la exploración completa sería exponencial respecto al número de posibles jugadas en cada nivel (r) y la profundidad de la solución (p) ($O(r^p)$). En cualquier juego real este valor es una cifra astronómica, por ejemplo, el factor de ramificación del ajedrez es aproximadamente de 35 y el del go esta alrededor de 300.

5.3 Seamos un poco más inteligentes

Es evidente que no podemos explorar todo el espacio de búsqueda para obtener el mejor conjunto de jugadas, así que debemos de echar mano del conocimiento que tenemos del juego para reducir la exploración.

Tendremos que introducir una función heurística que en este caso no medirá la bondad de un estado respecto a su distancia hasta una jugada ganadora. La filosofía de esta función heurística es diferente de las que hemos visto hasta ahora, ya que no existe ninguna noción de coste, ni tampoco de optimalidad, pues todas las jugadas ganadoras son igual de buenas. Esto hará que construir esta función sea todavía mas difícil que en las ocasiones anteriores, habrá que determinar cuál es la ventaja de cada jugador en cada estado³ y esto puede ser bastante difícil de determinar. También hemos de tener en cuenta que el coste de calcular la función heurística también influirá en el coste de la exploración.

Para este tipo de problemas, esta función nos podrá dar valores positivos y negativos, indicándonos qué jugador es el que tiene la ventaja. Consideraremos que si el valor es positivo la jugada es ventajosa para el jugador MAX, y si es negativo la ventaja es para MIN. Por convenio, las jugadas ganadoras de MAX tienen un valor de $+\infty$ y las ganadoras de MIN de $-\infty$.

³Nótese que la función heurística no solo evaluará los estados ganadores.

Además de utilizar una función heurística para guiar la búsqueda, usaremos una estrategia distinta para explorar el conjunto de jugadas. Ya que es imposible hacer una exploración exhaustiva, haremos una búsqueda en profundidad limitada, esto reducirá lo que podremos ver del espacio de búsqueda.

Decidiremos un nivel de profundidad máximo para la búsqueda, evaluaremos los estados que están en ese nivel y averiguaremos cuál es la mejor jugada a la que podemos acceder desde el estado actual. Hay que tener claro que con este procedimiento solo decidimos una jugada, y que repetimos la búsqueda en cada movimiento que tenemos que decidir. A pesar de que repetimos la búsqueda a cada paso, el número de nodos explorados es mucho menor que si exploráramos todo el árbol de búsqueda completo.

La calidad del juego vendrá determinada por la profundidad a la que llegamos en cada exploración. Cuanto más profunda sea la exploración mejor jugaremos, ya que veremos más porción del espacio de búsqueda. Evidentemente, cuanto más exploremos, más coste tendrá la búsqueda.

El algoritmo que realiza esta exploración recibe el nombre de **minimax**⁴ y es un recorrido en profundidad recursivo. El que sea un recorrido en profundidad hace que el coste espacial sea lineal, pero evidentemente el coste temporal será exponencial (dependerá de la profundidad máxima de exploración). El algoritmo 5.1 es su implementación.

El algoritmo tiene una función principal (**MiniMax**) que es la que retorna la mejor jugada que se puede realizar, y dos funciones recursivas mutuas (**valorMax** y **valorMin**) que determinan el valor de las jugadas dependiendo de si el nivel es de MAX o de MIN. La función **estado_terminal(g)** determina si el estado actual es una solución o pertenece al nivel máximo de exploración. La función **evaluacion(g)** calcula el valor de la función heurística para el estado actual.

5.4 Seamos aún más inteligentes

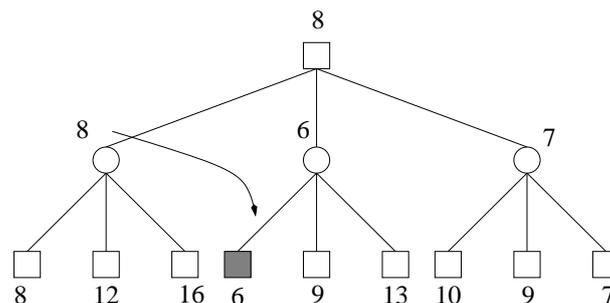
Un punto clave que hemos de tener en cuenta es que cuanto más profunda sea la exploración, mejor podremos tomar la decisión sobre qué jugada debemos escoger. Para juegos en los que el factor de ramificación sea muy grande, no podremos llegar a mucha profundidad, ya que el tiempo que necesitaremos para cada decisión será prohibitivo.

Para reducir este tiempo de exploración se han estudiado las propiedades que tienen estos árboles de búsqueda y se han desarrollado heurísticas que permiten no explorarlos en su totalidad y obtener el mismo resultado que obtendríamos con la exploración completa.

La heurística que veremos se denomina **poda alfa-beta** ($\alpha\beta$ **pruning**). Esta heurística aprovecha que el algoritmo del minimax hace una exploración en profundidad para guardar información sobre cuál es el valor de las mejores jugadas encontradas hasta cierto punto de la búsqueda para el jugador MAX y para el jugador MIN.

Lo que hace esta heurística es evitar seguir explorando nodos que sabemos que no van a variar la decisión que se va a tomar en niveles superiores, eliminándolos de la búsqueda y ahorrando tiempo.

Podemos ver por ejemplo el siguiente árbol de exploración:



⁴El nombre minimax proviene del teorema del Minimax del área de teoría de juegos enunciado por John von Neumann en 1928.

Algoritmo 5.1 Algoritmo minimax

Función: MiniMax (*g*)

movr:movimiento; max,maxc:entero

max $\leftarrow \infty$ **para cada** *mov* \in *mouv_posibles(g)* **hacer**| cmax \leftarrow valor_min(aplicar(*mov*,*g*))| **si** *cmax* > *max* **entonces**| | max \leftarrow *cmax*| | movr \leftarrow *mov*| **fin****fin****retorna** *movr***Función:** valorMax (*g*)

vmax:entero

si *estado_terminal(g)* **entonces**| **retorna** *valor(g)***sino**| vmax $\leftarrow -\infty$ | **para cada** *mov* \in *mouv_posibles(g)* **hacer**| | vmax \leftarrow **max**(vmax, valorMin(aplicar(*mov*,*g*)))| **fin**| **retorna** *vmax***fin****Función:** valorMin (*g*)

vmin:entero

si *estado_terminal(g)* **entonces**| **retorna** *value(g)***sino**| vmin $\leftarrow +\infty$ | **para cada** *mov* \in *mouv_posibles* **hacer**| | vmin \leftarrow **min**(vmin, valorMax(aplicar(*mov*,*g*)))| **fin**| **retorna** *vmin***fin**

Algoritmo 5.2 Algoritmo minimax con poda $\alpha\beta$

```

Función: valorMax (g, $\alpha$ , $\beta$ )
si estado_terminal(g) entonces
  | retorna valor(g)
sino
  | para cada mov  $\in$  movs_posibles(g) hacer
  |   |  $\alpha \leftarrow$  max( $\alpha$ ,valorMin(aplicar(mov,g) , $\alpha$ , $\beta$ ))
  |   | si  $\alpha \geq \beta$  entonces
  |   | | retorna  $\beta$ 
  |   | fin
  |   fin
  | retorna  $\alpha$ 
fin

```

```

Función: valorMin (g, $\alpha$ , $\beta$ )
si estado_terminal(g) entonces
  | retorna valor(g)
sino
  | para cada mov  $\in$  movs_posibles(g) hacer
  |   |  $\beta \leftarrow$  min( $\beta$ ,valorMax(apply(mov,g) , $\alpha$ , $\beta$ ))
  |   | si  $\alpha \geq \beta$  entonces
  |   | | retorna  $\alpha$ 
  |   | fin
  |   fin
  | retorna  $\beta$ 
fin

```

Podemos ver que el valor que propagaría el algoritmo del minimax a nodo raíz sería 8. Pero si nos fijamos en el nodo marcado, el segundo descendiente de la raíz ya sabe que el mejor nodo del primer descendiente tiene valor 8, y su primer descendiente vale 6. Dado que la raíz es un nodo MAX, este preferirá el valor 8 al 6. Dado que el segundo descendiente es un nodo MIN, éste siempre escogerá un valor que como máximo será 6. Esto nos hace pensar que, una vez hemos visto que el valor del nodo marcado es 6, el valor que salga de esa exploración no se elegirá (ya que tenemos un valor mejor de la exploración anterior). Esto lleva a la conclusión de que seguir explorando los nodos del segundo descendiente de la raíz no merece la pena, ya que el valor resultante ya no es elegible.

Podemos observar también que esta circunstancia se repite en el tercer descendiente al explorar su último nodo, pero al no quedar más descendientes esta heurística no nos ahorra nada. Esto querrá decir que la efectividad de esta heurística dependerá del orden de exploración de los nodos, pero desgraciadamente es algo que no se puede establecer a priori. En el caso medio, el coste asintótico del tiempo de exploración pasa de ser $O(r^p)$ a ser $O(r^{\frac{p}{2}})$.

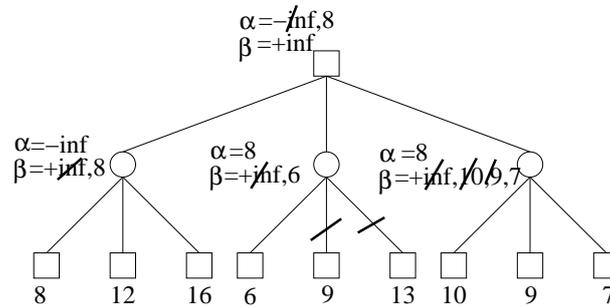
Esta heurística modifica el algoritmo del minimax introduciendo dos parámetros en las llamadas de las funciones, α y β , y representarán el límite del valor que pueden tomar las jugadas que exploremos. Una vez superado ese límite sabremos que podemos parar la exploración porque ya tenemos el valor de la mejor jugada accesible.

La función minimax se mantiene igual, solo varían las funciones que hacen la exploración de cada nivel, su código el del algoritmo 5.2. La llamada que hará la función MiniMax a la función valorMax le pasará $-\infty$ como valor de alfa y $+\infty$ como valor de beta.

En la función valorMax, alfa es el valor que se actualiza y beta se mantiene constante represen-

tando el valor de la mejor jugada encontrada hasta ahora. En la función `valorMin`, `beta` es el valor que se actualiza y `alfa` se mantiene constante representando el valor de la mejor jugada encontrada hasta ahora.

En la siguiente figura se ve como exploraría el algoritmo de minimax con poda alfa beta el ejemplo anterior:



La ventaja de utilizar la poda alfa beta es que podemos permitirnos ampliar el límite de profundidad de exploración, ya que nos ahorramos la exploración de parte del árbol de búsqueda, y así podemos conseguir mejores resultados.



Dado que la ordenación de los descendientes es crucial para obtener una poda efectiva se suelen utilizar heurísticas adicionales que permiten aproximar esa ordenación con el objetivo de aumentar las podas. Obviamente esto no es gratis, a veces es necesario repetir expansiones y memorizar nodos para obtenerlo.

Una estrategia habitual es utilizar un algoritmo de profundidad iterativa en lugar del de profundidad limitada para hacer una exploración por niveles. A cada nivel se pueden memorizar las evaluaciones que se obtienen para cada nodo y utilizar esa evaluación en la siguiente iteración. Esto permitirá que en la siguiente iteración se puedan ordenar los descendientes conocidos y explorar primero la rama más prometedora y poder podar con mayor probabilidad el resto de descendientes.

Como ya hemos visto el ratio de reexpansiones respecto a nodos nuevos para la profundidad iterativa tiende a cero con el aumento del factor de ramificación (en estos problemas suele ser muy grande), 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 $O(r^{\frac{p}{2}})$.

6.1 De variables y valores

Existen problemas específicos que, por sus propiedades, se pueden resolver más fácilmente utilizando algoritmos adaptados a ellos que utilizando algoritmos generales. Este es el caso de los problemas denominados de **satisfacción de restricciones** (**constraint satisfaction**¹).

Las características de un problema de satisfacción de restricciones son las siguientes:

- El problema se puede representar mediante un conjunto de variables.
- Cada variable tiene un dominio de valores (un conjunto finito de valores discretos o intervalos de valores continuos)
- Existen restricciones de consistencia entre las variables (binarias, n-arias)
- Una solución es una asignación de valores a esas variables

Esto quiere decir que todo problema de satisfacción de restricciones se puede definir a partir de una tripleta de elementos $\mathcal{P} = \langle X, D, R \rangle$, donde X es una lista de variables, D es una lista de los dominios que corresponden a las variables y R es una lista de relaciones entre conjuntos de variables que definen las restricciones que debe cumplir el problema.

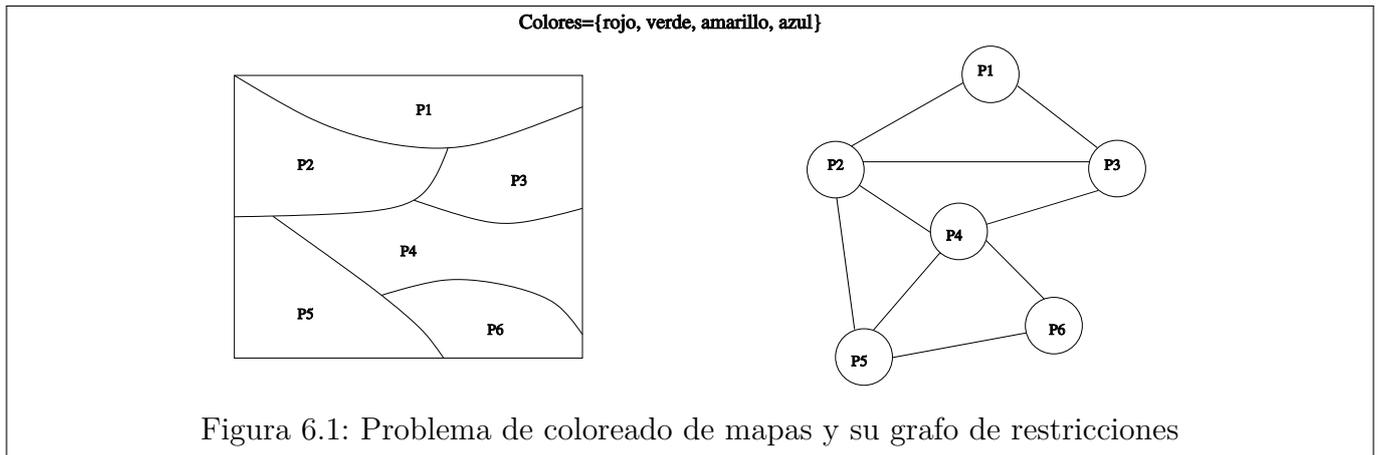
A partir de esta formulación genérica se pueden definir especializaciones y generalizaciones. Por ejemplo, si restringimos los dominios de las variables a valores binarios y las relaciones entre variables son disyunciones lógicas, tenemos problemas de satisfactibilidad (si restringimos el número de variables por relación a como mucho 3, tenemos el problema 3-SAT, un ejemplo típico de problema NP-completo). También podemos generalizar el problema admitiendo por ejemplo la posibilidad de añadir costes a las variables de manera que tengamos como objetivo optimizar su valor.

El interés de este tipo de problemas viene de su estrecha relación con otras áreas, como satisfactibilidad, álgebra relacional, teoría de base de datos, visión o gráficos y de que muchos problemas reales se pueden plantear como problemas de satisfacción de restricciones, como por ejemplo la planificación de tareas, logística (localización o asignación de recursos, personas, vehículos, ...), gestión de redes (electricidad, comunicaciones, ...), diseño/configuración, reconocimiento de formas en imágenes, razonamiento (temporal, espacial, ...), ... Estas aplicaciones hacen que sea un área bastante interesante desde un punto de vista práctico.

Todos los algoritmos de resolución de problemas de satisfacción de restricciones que veremos están pensados para el caso de que las restricciones sean binarias ya que se pueden representar mediante grafos (**grafos de restricciones**) en los que los nodos son las variables, las restricciones son los arcos, y los dominios de las variables son los diferentes valores que pueden tomar los nodos. Esto no supone ninguna limitación, dado que la mayor parte de los problemas con restricciones no binarias se pueden convertir en problemas con restricciones binarias introduciendo variables artificiales o transformando el grafo en un hipergrafo.

Ejemplo 6.1 *Un ejemplo clásico de problema de satisfacción de restricciones es el de coloreado de mapas (figura 6.1). El problema consiste en asignar colores a los países de un mapa de manera que dos países adyacentes no tengan el mismo color.*

¹En la literatura los encontrareis abreviados como CSP (constraint satisfaction problems).



El problema se puede representar como un grafo donde cada nodo es un país y los arcos entre nodos representan las fronteras comunes. Los nodos del grafo serían las variables del problema, los arcos representarían las restricciones entre pares de variables (sus valores han de ser diferentes) y los colores disponibles serían los dominios de las variables.

6.2 Buscando de manera diferente

Se puede plantear un problema de satisfacción de restricciones como un problema de búsqueda general. Dado que una solución es una asignación de valores a las variables del problema, solo haría falta enumerar de alguna manera todas las posibilidades hasta encontrar alguna que satisfaga las restricciones.

Evidentemente, esta aproximación es irrealizable ya que el conjunto de posibles asignaciones a explorar es muy grande. Suponiendo n variables y m posibles valores, tendríamos un espacio de búsqueda de $O(m^n)$.

Un problema adicional es que no hay ninguna noción de calidad de solución, ya que todas las asignaciones son igual de buenas y lo único que las diferencia 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.



Esto no es completamente cierto, ya que siempre podemos utilizar el número de restricciones que no se cumplen como función heurística. Desgraciadamente este número no suele ser muy informativo respecto a lo buenos o malos que pueden ser los vecinos de una no solución, ya que habrá no soluciones con pocas restricciones incumplidas, pero que hagan falta muchas modificaciones para hacerlas solución y al revés, soluciones que a pesar de incumplir varias restricciones, con unas pocas modificaciones se conviertan en solución.

Una ventaja con la que nos encontraremos es que, las propiedades que tienen estos problemas permiten derivar heurísticas que reducen el espacio de búsqueda de manera considerable.

Veremos tres aproximaciones a la resolución de problemas de satisfacción de restricciones, la primera se basará en búsqueda no informada, la segunda se basará en la reducción de espacio de búsqueda del problema y la tercera intentará combinar ambas.

6.2.1 Búsqueda con backtracking

La primera aproximación se basará en un algoritmo de búsqueda no informada. Plantearemos el problema de manera que podamos explorar el conjunto de posibles asignaciones de valores a las variables como una búsqueda en profundidad.

Podemos observar que, dada una asignación parcial de valores a las variables de un problema, ésta podría formar parte de una solución completa si no viola ninguna restricción, por lo tanto solo tendríamos que encontrar el resto de valores necesarios para completarla.

Esto nos hace pensar que podemos plantear la búsqueda más fácilmente si exploramos en el espacio de las soluciones parciales, que si lo planteamos como una búsqueda en el espacio de las soluciones completas como habíamos comentado antes. Esto nos puede indicar un algoritmo para hacer la exploración.

Podemos establecer un orden fijo para las variables y explorar de manera sistemática el espacio de soluciones parciales, asignando valores a las variables de manera consecutiva en el orden que hemos fijado. En el momento en el que una solución parcial viole alguna de las restricciones del problema, sabremos que no formará parte de ninguna solución completa, por lo que replantaremos las asignaciones que hemos realizado. A este tipo de búsqueda se la denomina búsqueda con **backtracking cronológico**. Se puede ver su implementación en el algoritmo 6.1.

Definiremos tres conjuntos con las variables del problema. Por un lado tendremos las variables pasadas, que serán las que ya tienen un valor asignado y forman parte de la solución parcial. Tendremos la variable actual, que será a la que debemos asignarle un valor. Y por último tendremos las variables futuras, que serán las que aún no tienen valor.

El algoritmo hace un recorrido en profundidad de manera recursiva por todas las asignaciones posibles de valores a cada una de las variables del problema. El parámetro `vfuturas` contiene todas las variables del problema en orden, el parámetro `solución` irá guardando las soluciones parciales. El hacer el recorrido en profundidad hace que el coste espacial de la búsqueda sea lineal.

El caso base de la recursividad es que el conjunto de variables futuras se haya agotado, el caso recursivo es iterar para todos los valores de la variable actual. El algoritmo continua las llamadas recursivas cuando consigue una asignación de valor a la variable actual que cumpla las restricciones del problema (`solucion.válida()`). En el caso de que ninguna asignación dé una solución parcial válida, el algoritmo permite que se cambie el valor de la variable anterior retornando la solución en un estado de fallo.

Ejemplo 6.2 *Otro problema clásico de satisfacción de restricciones es el de la N reinas que ya hemos comentado en el primer capítulo. Éste se puede plantear como un problema de satisfacción de restricciones en el que las reinas serían las variables, el dominio de valores sería la columna en la que se coloca la reina y las restricciones serían que las posiciones en las que colocamos las reinas no hagan que se maten entre sí.*

En la figura 6.2 podemos ver como exploraría el algoritmo de backtracking cronológico el problema en el caso de 4 reinas.

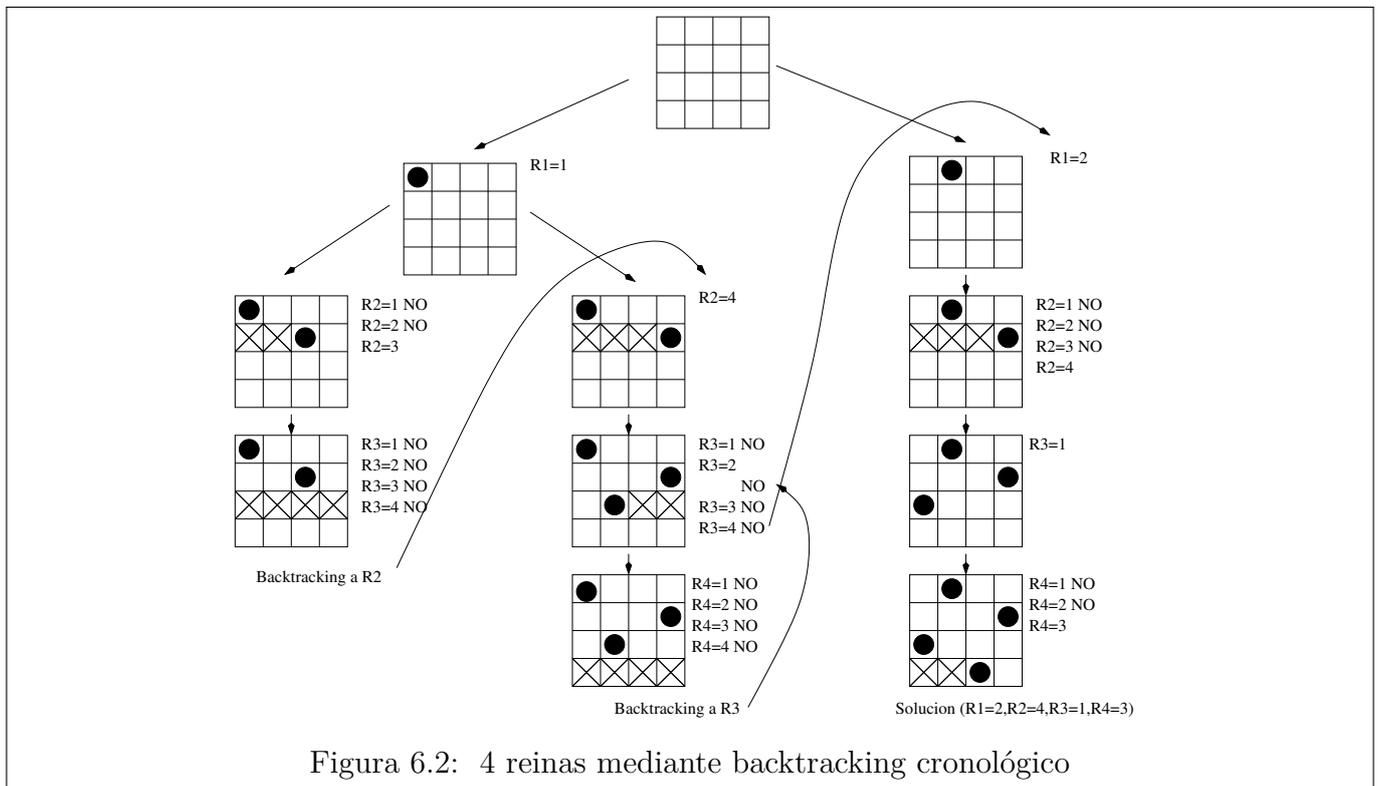
En el ejemplo se puede ver como el algoritmo va probando asignaciones para cada una de las reinas, avanzando en el momento que encuentra una asignación compatible con las asignaciones previas. En el caso de no haber ninguna asignación compatible, el algoritmo retrocede hasta la primera variable a la que le quedan valores por probar.

El algoritmo de backtracking cronológico puede hacer bastante trabajo innecesario a veces si la variable que provoca la vuelta atrás del algoritmo no es precisamente la última asignada, haciendo que se tengan que probar todas las combinaciones posibles de las variables que hay entre el primer punto donde se encuentra la violación de la restricción y la variable que la provoca. Esto ha hecho desarrollar variantes más inteligentes del backtracking cronológico, la más sencilla es la denominada **backjumping**, que intenta que el backtracking no sea a la variable anterior, sino que, teniendo en

Algoritmo 6.1 Algoritmo de backtracking cronológico

```

Función: backtracking_cronologico(vfuturas, solucion)
si vfuturas.es_vacio?() entonces
  | retorna solucion
sino
  vactual ← vfuturas.primer()
  vfuturas.borrar_primer()
  para cada v ∈ vactual.valores() hacer
    vactual.asignar(v)
    solucion.anadir(vactual)
    si solucion.valida() entonces
      solucion ← backtracking_cronologico(vfuturas,solucion)
      si no solucion.es_fallo?() entonces
        | retorna solucion
      sino
        | solucion.borrar(vactual)
    sino
      | solucion.borrar(vactual)
  retorna solucion.fallo()
  
```



Algoritmo 6.2 Algoritmo de arco-consistencia**Algoritmo:** Arco consistencia

```

R ← conjunto de arcos del problema /* ambos sentidos */
mientras se modifiquen los dominios de las variables hacer
    r ← extraer_arco(R)
    /* ri es la variable del origen del arco */
    /* rj es la variable del destino del arco */
    para cada v ∈ en el dominio de ri hacer
        si v no tiene ningún valor en el dominio de rj que cumpla r entonces
            borrar v del dominio de ri
            añadir todos los arcos que tengan como destino ri menos el (rj → ri)
        fin
    fin
fin

```

cuenta las restricciones involucradas, vuelva a la variable que tiene alguna restricción con la variable actual. Esto consigue evitar todas las pruebas de valores entre la variable actual y esa variable.

6.2.2 Propagación de restricciones

Otras vías a través de las que atacar este tipo de problemas son las denominadas técnicas de **propagación de restricciones**. Mediante éstas se pretende reducir el número de combinaciones que hay que probar, descartando todas aquellas que contengan asignaciones de valores que sabemos a priori que no pueden aparecer en ninguna solución.

Esta técnicas también sirven para reducir el número de vueltas atrás inútiles que realiza el algoritmo de backtracking cronológico como veremos en la siguiente sección, eliminando el mayor número posible de incompatibilidades entre los valores de las variables, de manera que no tengan que ser exploradas.

Estas técnicas se han desarrollado a partir de heurísticas derivadas de propiedades que tienen los grafos de restricciones. Las más utilizadas son las que se derivan de la propiedad denominada **k-consistencia**.

Se dice que un grafo de restricciones es **k-consistente** si para cada variable X_i del grafo, cada conjunto de k variables que la incluya y cada valor del dominio de X_i , 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.

Esta propiedad la podemos definir para cualquier valor de k, pero típicamente se utiliza el caso de k igual a 2, al que se denomina **arco-consistencia**. Diremos 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.

El algoritmo 6.2 que convierte un grafo de restricciones en arco-consistente. Este algoritmo es

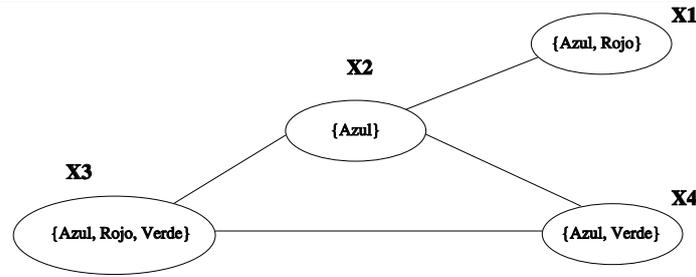


Figura 6.3: Ejemplo de coloreado de grafos

conocido como **AC3**, su coste temporal es $O(rd^3)$ y su coste espacial es $O(r)$, siendo r el número de restricciones del problema (contando los dos sentidos) y d es el tamaño de los dominios, existen algoritmos más eficientes, a costa de usar más espacio que tienen complejidad temporal $O(rd^2)$ y complejidad espacial $O(rd)$.

Ejemplo 6.3 En la figura 6.3 tenemos un problema de coloreado de grafos donde en cada uno de los nodos se indica el dominio de cada variable. El objetivo será colorear el grafo de manera que ningún nodo adyacente tenga el mismo color, por lo que cada arco es una restricción de desigualdad.

El conjunto de arcos con los que comenzaría el algoritmo son:

$$\{X_1 - X_2, X_2 - X_1, X_2 - X_3, X_3 - X_2, X_2 - X_4, X_4 - X_2, X_3 - X_4, X_4 - X_3\}$$

La ejecución del algoritmo sería la siguiente:

1. Seleccionamos $X_1 - X_2$ Eliminamos Azul del dominio de X_1
2. Seleccionamos $X_2 - X_1$ Todos los valores son consistentes
3. Seleccionamos $X_2 - X_3$ Todos los valores son consistentes
4. Seleccionamos $X_3 - X_2$ Eliminamos Azul del dominio de X_3
Tendríamos que añadir el arco $X_4 - X_3$
pero ya está
5. Seleccionamos $X_2 - X_4$ Todos los valores son consistentes
6. Seleccionamos $X_4 - X_2$ Eliminamos Azul del dominio de X_4
Tendríamos que añadir el arco $X_3 - X_4$
pero ya está
7. Seleccionamos $X_3 - X_4$ Eliminamos Verde del dominio de X_3
Añadimos el arco $X_2 - X_3$
8. Seleccionamos $X_4 - X_3$ Todos los valores son consistentes
9. Seleccionamos $X_2 - X_3$ Todos los valores son consistentes

Casualmente, al hacer arco-consistente el grafo hemos hallado la solución, pero no siempre tiene por que ser así.

La arco-consistencia no siempre nos asegura una reducción en los dominios del problema, por ejemplo, si consideramos el grafo del problema de las 4 reinas que hemos visto en el ejemplo 6.2, éste es arco-consistente, por lo que no se puede eliminar ninguno de los valores de las variables.

La k -consistencia para el caso de k igual a 3 se denomina **camino consistencia**, y permite eliminar más combinaciones, pero con un coste mayor. Existe la propiedad que se denomina **k-consistencia fuerte**, que se cumple cuando un grafo cumple la propiedad de consistencia desde 2 hasta k . Esta propiedad asegura que si el grafo tiene anchura² k y es k -consistente fuerte, encontraremos una solución sin necesidad de hacer backtracking, desgraciadamente probar esto necesita un tiempo muy elevado ($O(r^k d^k)$). Un resultado interesante es que, si el grafo de restricciones es un árbol, este se puede resolver sin backtracking si se convierte en arco consistente.

²Se puede calcular la anchura de un grafo en coste cuadrático.

Algoritmo 6.3 Algoritmo de forward checking

```

Función: forward checking (vfuturas, solución)
si vfuturas.es_vacio?() entonces
  | retorna solucion
sino
  | vactual ← vfuturas.primerero()
  | vfuturas.borrar_primerero()
  | para cada v ∈ vactual.valores() hacer
  |   | vactual.asignar(v)
  |   | solucion.anadir(vactual)
  |   | vfuturas.propagar_restricciones(vactual) /* forward checking */
  |   | si no vfuturas.algun_dominio_vacio?() entonces
  |   |   | solucion ← forward_checking(vfuturas, solucion)
  |   |   | si no solucion.es_fallo?() entonces
  |   |   |   | retorna solucion
  |   |   |   | sino
  |   |   |   |   | solucion.borrar(vactual)
  |   |   |   | sino
  |   |   |   |   | solucion.borrar(vactual)
  |   |   |   | retorna solucion.fallo()
  |   |   | retorna solucion.fallo()
  |   | retorna solucion.fallo()

```

6.2.3 Combinando búsqueda y propagación

La combinación de la búsqueda con backtracking y la propagación de restricciones es la que da resultados más exitosos en estos problemas. La idea consiste en que cada vez que asignemos un valor a una variable realicemos la propagación de las restricciones que induce esa asignación, eliminando de esta manera valores de las variables que quedan por asignar. Si con esta propagación, alguna de las variables se queda sin valores, deberemos hacer backtracking.

La ventaja de este método es que podremos hacer backtracking antes de que llegemos a la variable que acabará provocándolo. De esta manera nos ahorraremos todos los pasos intermedios.

Existen diferentes algoritmos que combinan búsqueda y propagación, dependiendo de que tipo de consistencia se evalúe en cada paso de búsqueda. El algoritmo más sencillo es el denominado de **forward checking**.

Este algoritmo añade a cada paso de asignación de variables una prueba de arco-consistencia entre las variables futuras y la variable actual. Esta prueba de arco consistencia consigue que todos los valores de las variables futuras que no sean consistentes con la asignación a la variable actual sean eliminados, de manera que si algún dominio de alguna variable futura queda vacío podemos reconsiderar la asignación actual o hacer backtracking.

El algoritmo es muy parecido al que hemos visto para el backtracking cronológico, ahora incluiríamos la reducción de dominios mediante la prueba de arco consistencia tras hacer la asignación de valor a la variable actual (*vfuturas.propagar_restricciones(vactual)*) y la comprobación de la validez de la solución parcial se haría mediante la comprobación de que todas las variables futuras tienen aún valores (*vfuturas.algun_dominio_vacio?()*). La implementación se puede ver en el algoritmo 6.3.

Ejemplo 6.4 En la figura 6.4 podemos ver de nuevo el ejemplo 6.2 utilizando el algoritmo de forward checking. Se puede comprobar que el número de asignaciones que se realizan es menor, ya que los

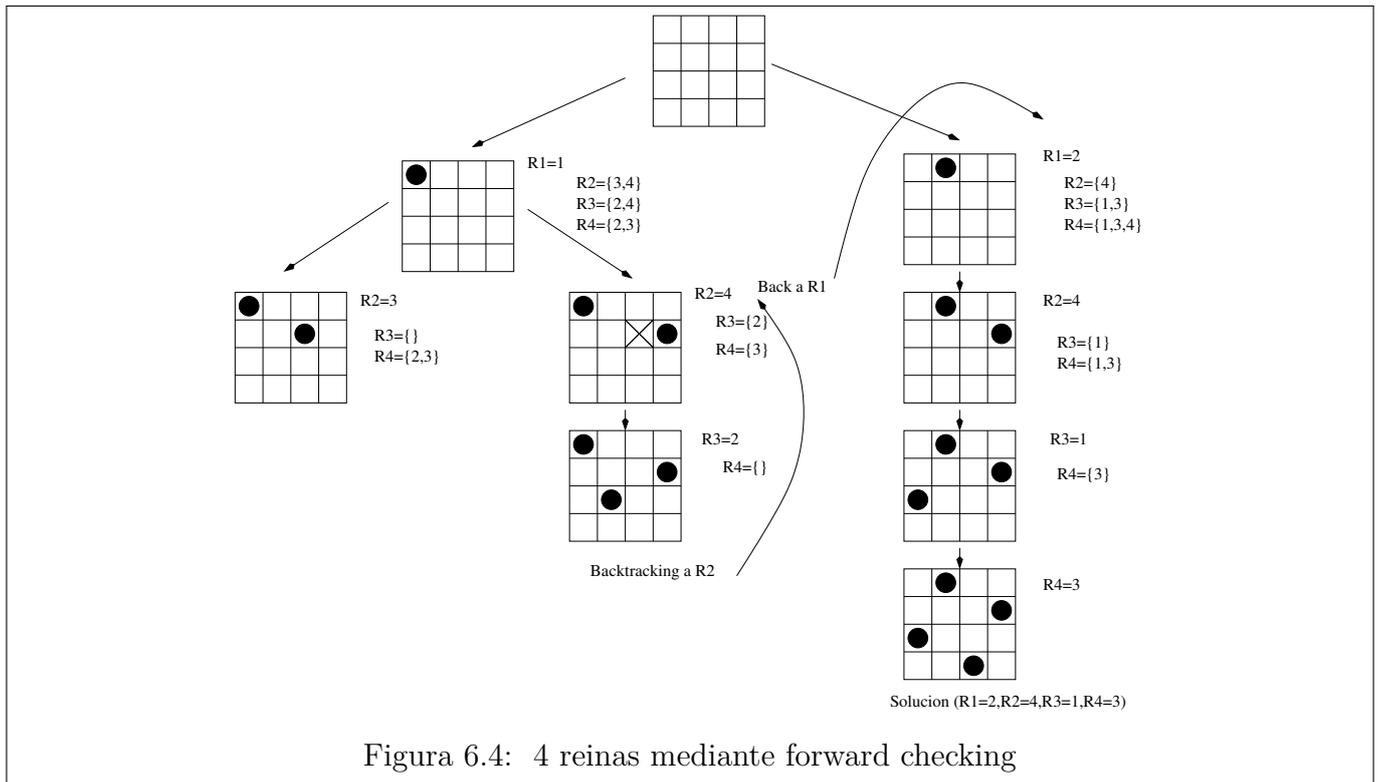


Figura 6.4: 4 reinas mediante forward checking

valores no compatibles son eliminados por las pruebas de consistencia. Ahora el coste ha pasado a las pruebas de consistencia que se realizan en cada iteración.

En este caso a cada paso el algoritmo asigna un valor a la reina actual y elimina de los dominios de las reinas futuras las asignaciones que son incompatibles con la actual. Esto nos permite hacer backtracking antes que en el ejemplo previo, reduciendo el número de asignaciones exploradas.

Hay que tener en cuenta que la prueba de arco consistencia de cada iteración incluye la comprobación de los valores que quedan en las variables futuras, por lo que dependiendo del problema el ahorro puede no existir.

Existen algoritmos que realizan pruebas de consistencia mas fuertes, como por ejemplo el algoritmo **RFL (Real Full Lookahead)** que además comprueba la arco-consistencia entre todas las variables futuras, o el algoritmo **MAC (Maintaining Arc Consistency)** que convierte el problema en arco consistente a cada paso.

Evidentemente, esta reducción de los dominios de las variables no sale gratis, el número de comprobaciones a cada paso incrementa el coste por iteración del algoritmo. Dependiendo del problema, el coste de propagación puede hacer que el coste de hallar una solución sea mayor que utilizar el algoritmo de backtracking cronológico.

6.3 Otra vuelta de tuerca

Muchas veces estas heurísticas no son suficientes para reducir el coste de encontrar una solución, por lo que se combinan con otras que también se han mostrado efectivas en este tipo de problemas.

Hasta ahora siempre hemos mantenido un orden fijo en la asignación de las variables, pero este puede no ser el más adecuado a la hora de realizar la exploración. Se utilizan diferentes heurísticas de ordenación dinámica de variables de manera que siempre se escoja la que tenga más probabilidad de llevarnos antes a darnos cuenta de si una solución parcial no nos llevará a una solución.

Las heurísticas más utilizadas son:

- **Dominios mínimos (Dynamic value ordering)**, escogemos la variable con el menor número de valores.
- **Min width ordering**, escogemos la variable conectada al menor número de variables no instanciadas.
- **Max degree ordering**, escogemos la variable más conectada en el grafo original.

No es despreciable el ahorro que se puede obtener mediante este tipo de heurísticas, por ejemplo, para el problema de las 20 reinas, hallar la primera solución con backtracking cronológico necesita alrededor de $2.5 \cdot 10^7$ asignaciones, utilizando forward checking se reduce a alrededor de $2.4 \cdot 10^6$ asignaciones, y si utilizamos la heurística de dominios mínimos combinada con el forward checking hacen falta alrededor de $4 \cdot 10^3$ asignaciones.

6.4 Ejemplo: El Sudoku

Muchos de los rompecabezas que aparecen habitualmente en la sección de pasatiempos son problemas de satisfacción de restricciones. Esto quiere decir que se pueden formular como un conjunto de variables a las que les corresponde un dominio de valores y un conjunto de restricciones sobre ellas.

Uno de estos rompecabezas es el sudoku (figura 6.5) para este problema el espacio de búsqueda es de $9!^9 \approx 2.3 \times 10^{33}$ posibles tableros, de los que aproximadamente 6.7×10^{24} son solución. En este caso tenemos una matriz de 9×9 , lo que nos da un total de 81 variables, cada variable puede contener un número del 1 a 9. Las restricciones aparecen primero entre las variables según las filas y columnas, de manera que para una fila o columna un número puede aparecer solamente una vez y además hay una restricción adicional entre las variables de las nueve submatrices que forman el problema, dentro de cada submatriz no puede haber dos variables con el mismo número.

En el caso de este problema, el entretenimiento no viene de encontrar una solución, sino de obtener una solución dado que hemos dado ya ciertos valores a algunas de las variables. La dificultad de encontrar la solución se puede variar estableciendo el número de variables a las que damos el valor. 1995.

1	2	3	4	5	6	7	8	9
4	5	6	7	8	9	1	2	3
7	8	9	1	2	3	4	5	6
2	3	1	5	6	4	8	9	7
5	6	4	8	9	7	2	3	1
8	9	7	2	3	1	5	6	4
3	1	2	9	7	8	6	4	5
6	4	5	3	1	2	9	7	8
9	7	8	6	4	5	3	1	2

Figura 6.5: Una solución al juego del sudoku

A*, 24
Algoritmo genético canónico, 45
Algoritmo minimax, 50
Algoritmos genéticos, 42
arco-consistencia, 59

Búsqueda en anchura, 13
Búsqueda en profundidad, 14
Búsqueda en profundidad iterativa, 16
Backtracking cronológico, 57
Beam search, 38
Branch and bound, 36
Búsqueda en haces, 38

Constraint satisfaction, 55

Espacio de estados, 6
Estado, 6
Estrategia de enfriamiento, 40

Forward checking, 61
Función de adaptación, 42
Función de fitness, 42

greedy best first, 24

Hill-climbing, 36

IDA*, 30
Iterative Deepening, 16

k-consistencia, 59

Memory bound A*, 32

One point crossover, 44
Operador, 6
Operador genético de cruce, 43
Operador genético de mutación, 43

Población, 44
poda alfa-beta, 50
Propagación de restricciones, 59

Random restarting hill climbing, 38
Recursive best first, 31
Relación de accesibilidad, 6

Satisfacción de restricciones, 55
Simulated annealing, 40
Solución, 6
Steepest ascent hill climbing, 37