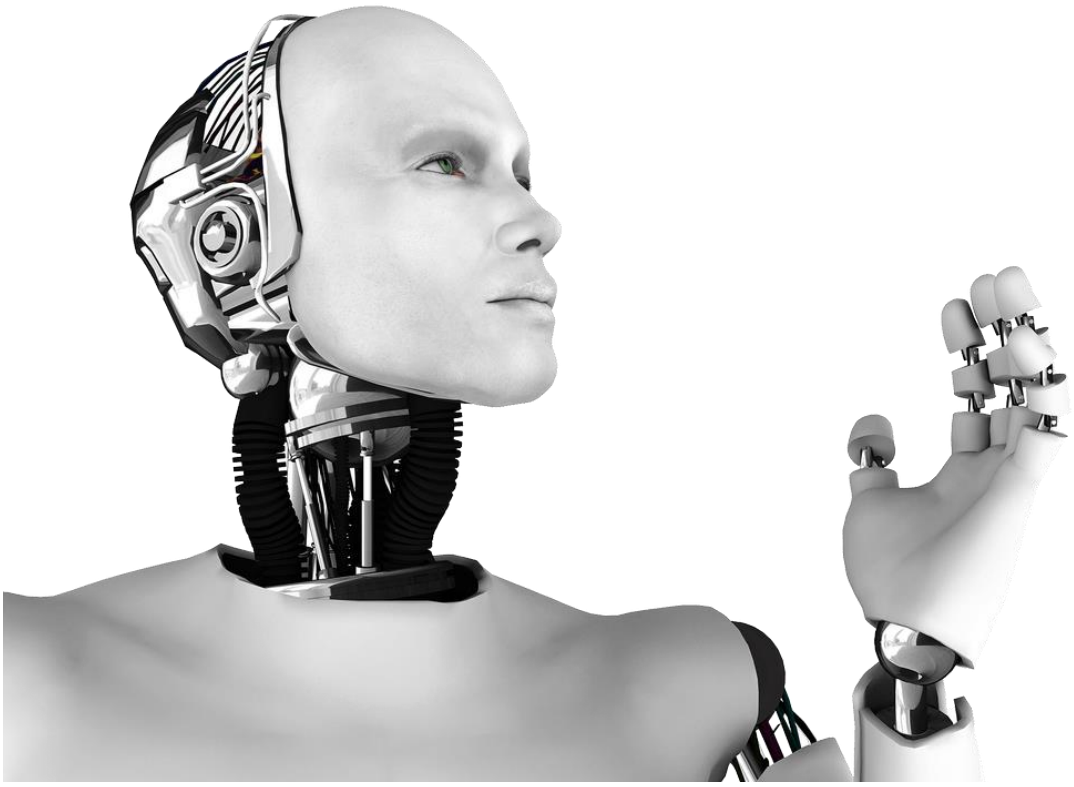


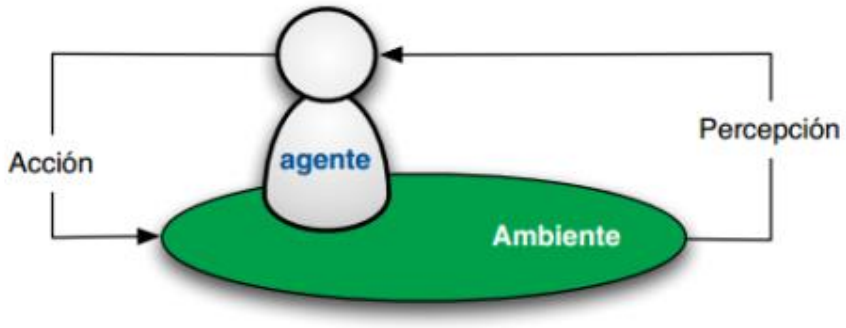
PRIMER PARCIAL DE IA

Posibles preguntas

Jose Antonio Padial Molina

1. El concepto de Agente. Agentes Racionales vs. Agentes Inteligentes. Arquitecturas de Agentes.

Al igual que ocurre con la definición de la IA, se puede encontrar un gran número de conceptos de agente, siendo la de Russell y Norvig 2008 la más sencilla, que considera un agente cono una entidad que percibe y actúa sobre un entorno. Basándose en esta definición, se puede caracterizar distintos agentes de acuerdo con los atributos que posean y que definen su comportamiento para resolver un determinado problema.



Entendemos por **agente** algo que es capaz de razonar, y por **agente racional,** aquel que actúa con la intención de alcanzar el mejor resultado o, cuando hay incertidumbre, el mejor resultado esperado.
Particularizando sobre lo que nos concierne, los agentes informáticos deben encontrarse dotados de características que aporten una mayor operatividad respecto a los "programas" convencionales, ya sea mediante atributos que los distingan, controles autónomos de percepción de su entorno y adaptación a éste ante cambios que puedan acontecer y repercutan de algún modo en su actuación, mediante una duración prolongada de su funcionalidad y de una persistencia mayor en los resultados generados en el tiempo, y por último, de una fiabilidad completa en la consecución de los objetivos marcados.
La práctica de uso de las teorías que representan a la IA desde el enfoque de diseño de un agente racional entraña notables beneficios, siendo reseñable el afirmar que garantiza una mayor proximidad a la racionalidad humana que bajo sólo las leyes del pensamiento, o que aporta una mayor afinidad al apoyarse en fines científicos, en constante evolución, frente a enfoques basados en la conducta o pensamiento humano, claramente definidos, de aplicación general.

1. Arquitecturas Deliberativas

Los agentes con una arquitectura deliberativa se caracterizan porque contienen explícitamente un *modelo simbólico del entorno*, y las decisiones (por ejemplo, sobre qué acciones realizar) son tomadas vía razonamiento lógico (o al menos pseudo-lógico) basado en equiparación de patrones y manipulación simbólica. Los agentes deliberativos mantienen la tradición de la Inteligencia Artificial clásica, basada en la hipótesis de los sistemas de símbolos físicos enunciada por Newell y Simon [Newell & Simon, 72].

Además de una representación simbólica del entorno, los agentes deliberativos tienen capacidad para llevar a cabo decisiones lógicas utilizando el conocimiento que con el que cuentan y modificando su estado interno, que frecuentemente de denomina *estado mental*.

Lo más común es que el estado mental de los agentes deliberativos, que normalmente son llamados *agentes BDI* (belief, desire, intention), esté formado por cinco componentes:

- *Creencias*: conocimiento que el agente tiene de si mismo y de su entorno, que puede ser incompleta o incorrecta.
- *Deseos*: objetivos que el agente desea cumplir a largo plazo
- *Metas*: subconjunto de los deseos que el agente puede conseguir. Han se ser realistas y no tener conflictos entre ellas.
- *Intenciones*: subconjunto de las metas que el agente intenta conseguir.
- *Planes*: combinan las intenciones en unidades consistentes. Las intenciones constituyen subplanes del plan global del agente, de manera que el conjunto de todos los planes refleja las intenciones del agente.

La construcción de agentes con arquitectura deliberativa presenta principalmente dos problemas [Brenner, 98]:

- El modelado del entorno y la elección de un lenguaje de representación simbólico adecuado y práctico puede crear muchas dificultades y supone un gran esfuerzo.
- Actualizar a tiempo todo el conocimiento necesario también ocasiona dificultades, ya que normalmente el conocimiento y los recursos necesarios no están disponibles en un tiempo limitado. Si además el

entorno en el que se encuentra el agente es dinámico y por lo tanto cambia relativamente rápido, el problema se acrecenta aún más, ya que si la visión que tiene agente sobre el mundo no es actual no puede ser muy proactivo.

Es decir, para construir un agente deliberativo el modelo simbólico del mundo deberá ser lo suficientemente complejo como para proporcionar al agente el conocimiento que necesita para realizar razonamientos lógicos, pero por otro lado deberá ser lo suficientemente simple como para que el sistema pueda ser actualizado a tiempo. Por otro lado, las teorías BDI proporcionan un modelo conceptual claro de conocimiento, metas y cometidos de un agente, pero deben de ser extendidas para soportar el diseño en aplicaciones prácticas de agentes limitados por recursos y de agentes dirigidos a metas.

2. Arquitecturas Reactivas

Los agentes reactivos se caracterizan porque pueden operar rápida y efectivamente sin la necesidad de procesar una representación simbólica del entorno, ya que representan una categoría de agentes que no posee modelos simbólicos externos del entorno en el que se encuentran [Nwana, 96]. Los agentes reactivos toman decisiones basadas totalmente en el presente, sin hacer uso de lo que ha ocurrido en el pasado, porque no conocen su historia [Weiss, 99]. Este tipo de agentes actúan siguiendo un esquema *estímulo-respuesta* según el estado actual del entorno en el que están embebidos. La característica más importante de los agentes reactivos es el hecho de que los agentes son relativamente simples e interaccionan con otros agentes de manera sencilla. Sin embargo, si se observa de una forma global el conjunto de agentes, las interacciones pueden dan lugar a patrones de comportamiento muy complejos.

Por el hecho de ser agentes sencillos, la capacidad de realizar razonamientos complejos desaparece en su mayor parte, y al contrario que los agentes deliberativos la inteligencia que puedan mostrar no proviene de modelos internos, sino de la interacción con su entorno [Brenner, 98].

Un agente reactivo está formado por los siguientes tipos de módulos:

- *Módulos de interacción con el entorno (sensores y actuadores* en la **¡Error! No se encuentra el origen de la referencia.** *).*
- *Módulos de competencia* (cada una de la capas en la **¡Error! No se encuentra el origen de la referencia.**).

Los *módulos de competencia* se desarrollan a partir de un conjunto de dependencias que el agente reconoce observando su entorno. Estos módulos son capaces de chequear continuamente el entorno, identificar una situación específica que está ocurriendo en él e iniciar una reacción directa ante esa situación.

La mayoría de las arquitecturas reactivas están basadas en reglas *situación-acción*, cuyo funcionamiento es similar al de una base de reglas. Así los agentes actúan de forma apropiada según su situación, donde una *situación* es una combinación de eventos internos y externos potencialmente compleja [Connah, 94]. Los agentes situación-acción se han utilizado en PENGI, un juego de ordenador [Agre, 87]. Los investigadores de los laboratorios Philips en Redhill, UK, han implementado un lenguaje situación-acción llamado RTA [Grahan & Wavish, 91] y lo han utilizado para implementar caracteres en juegos de computador. [Kaebling & Rosenschein, 91] han propuesto otro lenguaje que utiliza lógica modal que está basado en un paradigma llamado *situated automata*, en el que un agente se especifica en términos declarativos.

El área de aplicación usual de los agentes sotfware reactivos es el mundo de los juegos o la industria del entretenimiento y robots.

De forma resumida, las arquitecturas reactivas presentan las siguientes ventajas [Weiss, 99] :

- Simplicidad.
- Economía.
- Eficiencia computacional.
- Robustez ante fallos

Y los siguientes inconvenientes:

- Es necesaria una gran cantidad de información local.
- El aprendizaje es problemático.
- La construcción de agentes reactivos requiere experimentación, por lo que supone una gran cantidad de tiempo.
- La construcción de sistemas grandes es imposible.

- Los sistemas reactivos únicamente pueden ser utilizados para su propósito original.
- Existen pocas aplicaciones basadas en estas arquitecturas.

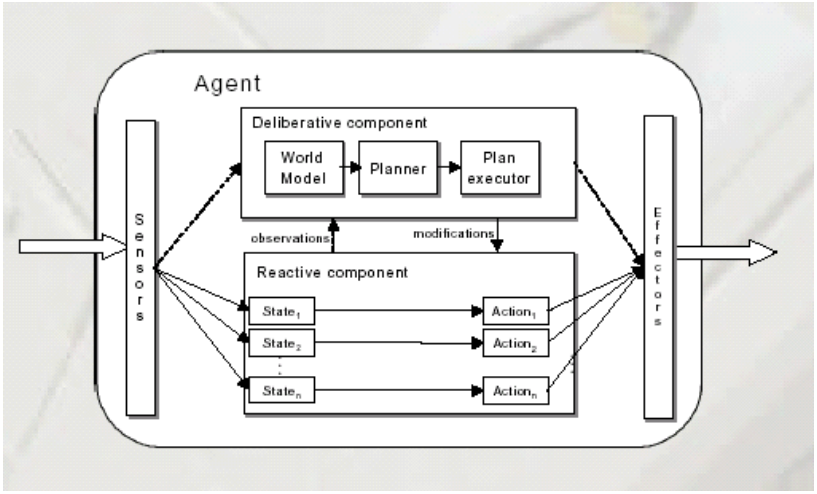
La crítica principal que han tenido las arquitecturas reactivas es que son excesivamente rígidas y simples como para producir comportamientos inteligentes como planificación o aprendizaje.

3. Arquitecturas Híbridas

Las arquitecturas híbridas combinan componentes de tipo reactivo con componentes de tipo deliberativo, como se observa en la **¡Error! No se encuentra el origen de la referencia.**. La parte reactiva interacciona con el entorno y reacciona rápidamente a los eventos que en él se producen sin invertir tiempo en realizar razonamiento, mientras que la parte deliberativa planifica y se encarga de la parte de toma de decisiones, es decir, realiza tareas a un nivel de abstracción superior.

Las arquitecturas híbridas pretenden aprovechar los beneficios que proporcionan las arquitecturas reactivas y los beneficios que proporcionan las deliberativas, ya que para la mayoría de los problemas no es adecuada ni una arquitectura puramente deliberativa ni una arquitectura puramente reactiva.

Típicamente los sistemas híbridos se diseñan siguiendo una arquitectura jerárquica en capas, en la que las capas más bajas son principalmente reactivas y las capas más altas son principalmente deliberativas.



Aunque las arquitecturas híbridas como Touring Machines [Ferguson, 92], InterRRaP [Müller, 93] y COSY [Burmeister & Sundermeyer, 92] tienen algunas ventajas sobre las puramente deliberativas y las puramente reactivas, tienen la dificultad potencial de que tienden a ser *ad-hoc* y aunque sus estructuras están bien motivadas desde el punto de vista de diseño, no está claro que sean motivadas por una teoría muy formal, o en general no se suele especificar una teoría que las soporte. En particular, arquitecturas Touring Machines, que contienen un número de capas independientes que compiten unas con otras en tiempo real para controlar la actividad del agente parecen estar faltas de formalización Lo que no esta claro es que la falta de formalización sea una desventaja seria ya que al menos se cuenta con un buen modelo para un agente particular, a pesar de que sea muy dificultoso generalizar y reproducir sus resultados en dominios distintos.

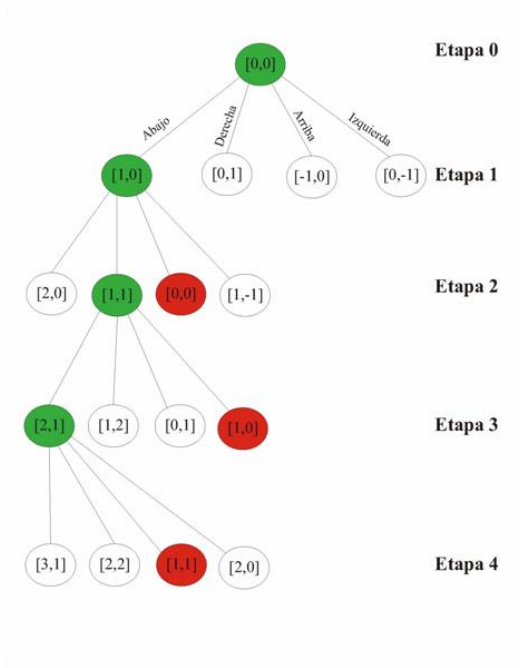
2. Características de los Agentes reactivos y deliberativos. Similitudes y diferencias. Arquitecturas.

Respondida en la 1.

3. Métodos de búsqueda no informada.

Existen, básicamente, dos estrategias de recorrido de un espacio de búsqueda, en **anchura** y en **profundidad**. Los algoritmos que veremos a continuación se basan en una de las dos (o incluso en las dos). El problema principal que tienen es que, al ser exhaustivos y sistemáticos su coste puede ser prohibitivo para la mayoría de los problemas reales, por lo tanto, solo serán aplicables en problemas pequeños, pero presentan la ventaja de que no es necesario ningún conocimiento adicional sobre el problema, por lo que siempre son aplicables.

Búsqueda genérica (o sin estrategia)



Los espacios de búsqueda, en general, pueden ser representados como árboles o como grafos. La diferencia principal entre los algoritmos que encontraremos para unos u otros radica, fundamentalmente, en que los algoritmos para árboles no necesitan mantener una lista de los nodos por los que ya ha pasado, ya que no pueden volver a visitarse nodos en el recorrido de búsqueda por el árbol. Sin embargo, si trabajamos con grafos podemos encontrar caminos que repitan nodos, y en este caso necesitaremos de una memoria de almacenamiento que nos indique si ya hemos visitado ese nodo en una etapa anterior o no.

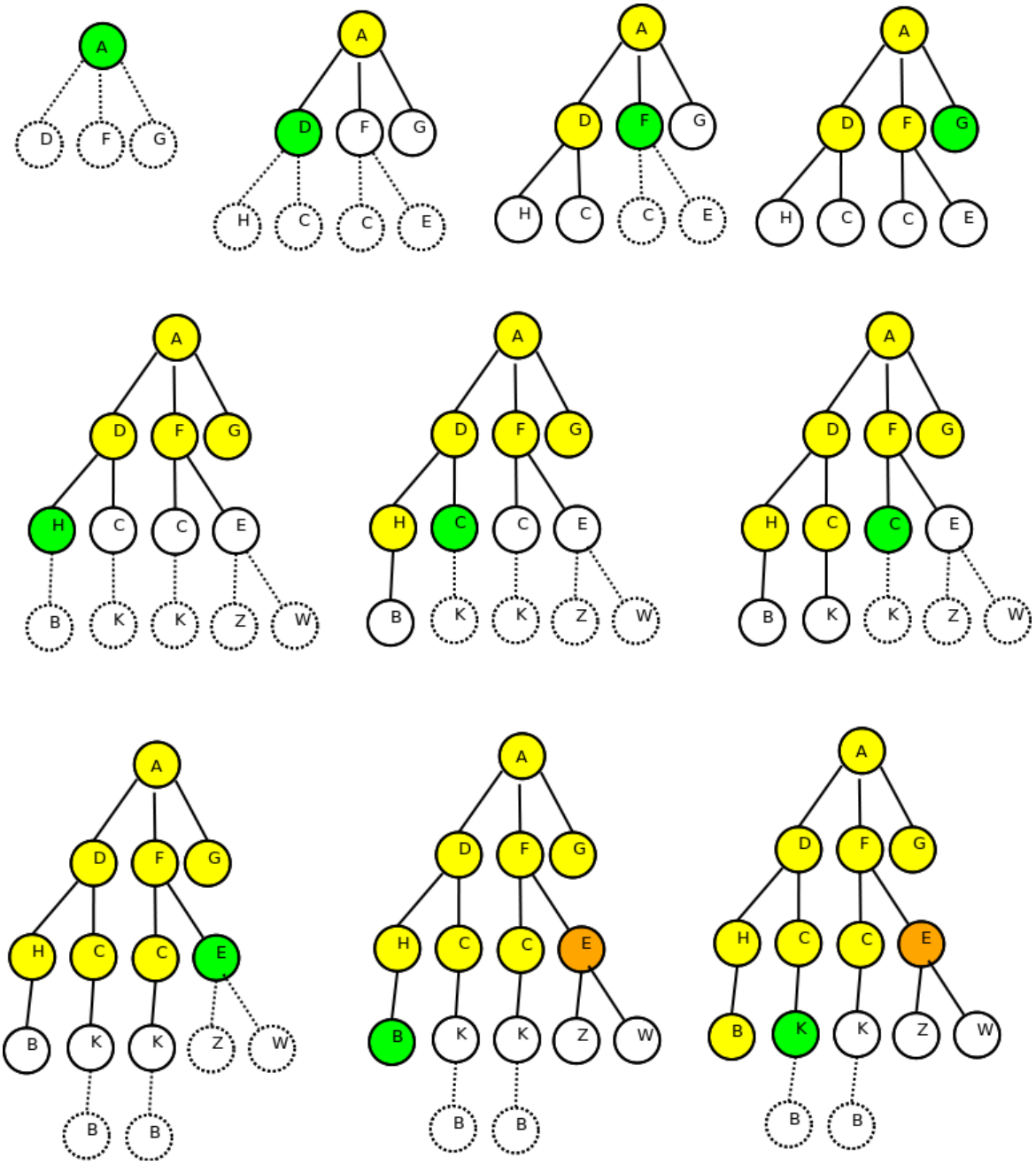
Durante una búsqueda, la **frontera** es la colección de nodos que esperan ser visitados (habitualmente almacenados en forma de pila, cola, lista...). Los nodos de la frontera se dicen **abiertos**. Cuando un nodo se elimina de la frontera, se etiqueta como **cerrado**.

En el algoritmo de búsqueda genérica no se proporciona ninguna estrategia para el recorrido de los nodos, simplemente es un patrón que indica que "existe una forma" (no determinada) de construir los caminos que empiezan en el nodo raíz y acaban en las hojas. Sobre este algoritmo general, cada uno de los que veremos a continuación añade únicamente una estrategia específica para decidir cómo se construyen dichos caminos. Dejaremos los algoritmos para grafos para cuando introduzcamos

heurísticas, ya que los que veremos aquí sacan provecho del hecho de que trabajamos con árboles y se comportan muy mal (hasta el punto de poder ser inútiles) cuando los aplicamos sobre grafos.

Búsqueda en anchura

La idea principal de la **Búsqueda en Anchura** (**BFS**) consiste en visitar todos los nodos que hay a profundidad iiantes de pasar a visitar aquellos que hay a profundidad i+1i+1. Es decir, tras visitar un nodo, pasamos a visitar a sus hermanos antes que a sus hijos. Si usamos estructuras de programación habituales, una posible implementación para BFS se haría almacenando el conjunto de nodos abiertos como una cola, a la que se accede por un procedimiento FIFO (el primero que entra es el primero que sale). Cuando hagamos uso de estructuras de programación basadas en agentes veremos que la implementación de este algoritmo es ligeramente distinta, y mantienen el almacenamiento haciendo uso de la propia estructura que existe entre los agentes involucrados.



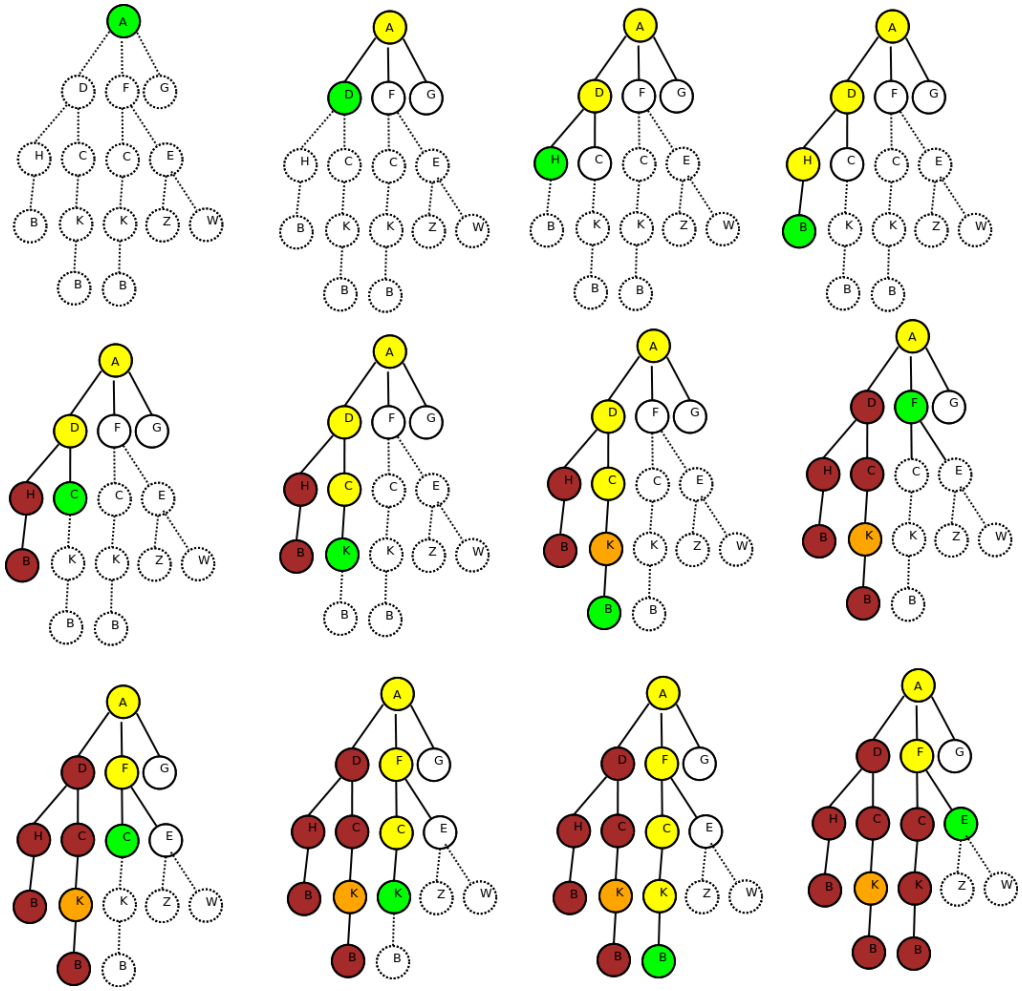
Este algoritmo es **completo**, es decir, si existe solución, este algoritmo la encuentra. Más aún, es **óptimo**, en el sentido de que si hay solución, encuentra una de las soluciones a distancia mínima de la raíz.

Respecto al tratamiento de nodos repetidos, se comporta bien. Si el nodo generado actual ya apareció en niveles superiores (más cerca de la raíz), el coste actual será peor ya que su camino desde la raíz es más largo, y si está al mismo nivel, su coste será el mismo. Esto quiere decir que si nos encontramos un nodo que ya ha sido repetido, su coste será peor o igual que algún nodo anterior visitado o no, de manera que lo podremos

descartar, porque o lo hemos expandido ya o lo haremos próximamente con mejor coste.

Búsqueda en Profundidad

Al igual que en el caso de la búsqueda en anchura, la **búsqueda en profundidad** (**DFS**) también puede ser vista como un proceso por niveles, pero con la diferencia de que, tras visitar un nodo, se visitan sus hijos antes que sus hermanos, por lo que el algoritmo tiende a bajar por las ramas del árbol hacia las hojas antes de visitar cada una de las ramas posibles. De nuevo, si hacemos uso de estructuras clásicas de programación, DFS se puede implementar por medio de una pila accediendo a sus elementos por un proceso de LIFO (último en entrar, primero en salir).



Como en el caso del BFS, la complejidad en tiempo del DFS es del orden de bdbd. Es exponencial ya que en el peor caso DFS tiene que visitar todos

los nodos. Sin embargo, la complejidad en espacio es lineal en dd, donde dd es la longitud del camino más largo posible, ya que el máximo número de nodos que se almacenan es del orden bdbd.

DFS no es ni óptimo ni completo. No es óptimo porque si existe más de una solución, podría encontrar la primera que estuviese a un nivel de profundidad mayor, y para ver que no es completo es necesario irse a ejemplos en los que el espacio de búsqueda fuese infinito: *supongamos un robot que puede moverse a izquierda o derecha en cada paso y ha de encontrar un objeto; la búsqueda en profundidad le obligaría a moverse indefinidamente en una sola dirección, cuando el objeto podría estar en la dirección contraria (e incluso a un solo paso del origen)*. Para evitar este problema es común trabajar con una pequeña variante de este algoritmo que se llama de **Profundidad limitada**, que impone un límite máximo al nivel alcanzado.

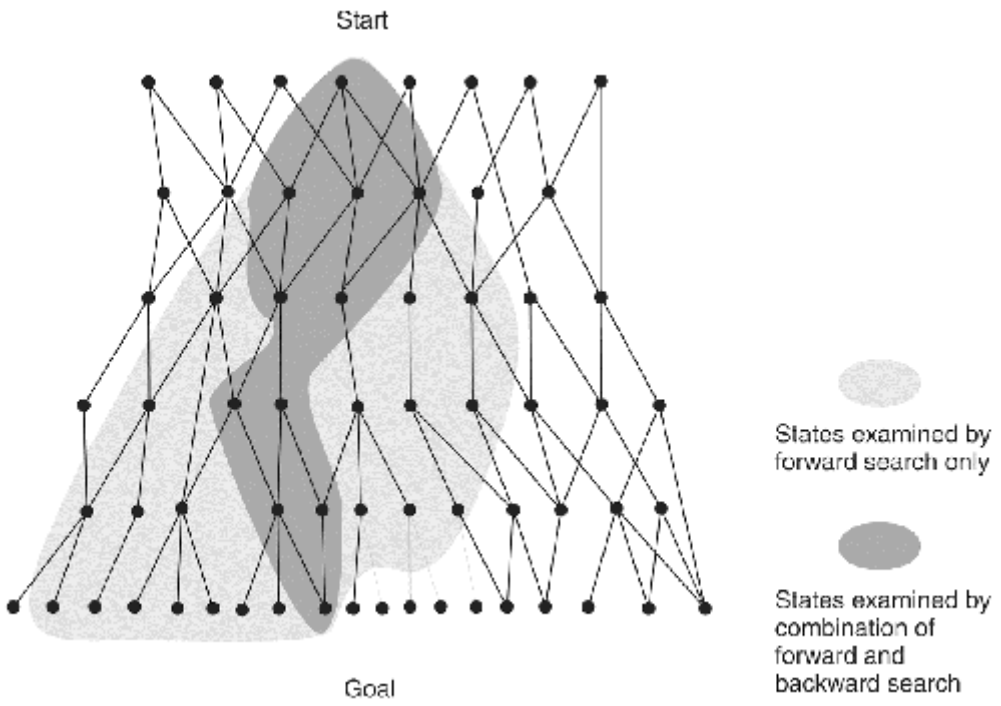
En el caso de la búsqueda en profundidad, el tratamiento de nodos repetidos no es crucial ya que al tener un límite en profundidad los ciclos no llevan a caminos infinitos. No obstante, en este caso se pueden comprobar los nodos en el camino actual, ya que está completo en la estructura de nodos abiertos. 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, y permite obtener soluciones que se encuentren a gran profundidad.

Búsqueda Bidireccional

Un problema de **Búsqueda bidireccional** es una extensión del problema de búsqueda general planteado antes. Se define como una 5-tupla (X,S,G,δ,γ)(X,S,G,δ,γ), donde (X,S,G,δ)(X,S,G,δ) es un problema de búsqueda básico y γ:X→2Xγ:X→2X es una función de **transición inversa**, es decir, que γ(x)γ(x) es el conjunto de predecesores de xx.

En la búsqueda bidireccional (BB), tal y como sugiere su nombre, se realizan dos búsquedas simultáneas: una desde el estado inicial hasta el estado final, y otra desde el estado final hasta el estado inicial. La búsqueda global acaba cuando ambas búsquedas parciales se encuentran. Sin embargo, no todos

los problemas de búsqueda básicos se pueden plantear de forma sencilla como problemas de búsqueda bidireccionales, muchas veces porque no es sencillo proporcionar la función de transición inversa.



Habitualmente, se suele implementar como un par de algoritmos BFS simultáneos, así que, como ellos, será óptimo y completo. Sin embargo, la BB mejora a BFS en términos de complejidad, ya que su complejidad en tiempo es exponencial en $d/2$ (cuando ambas búsquedas deben recorrer todos los nodos que le corresponden). Su complejidad en espacio es también del orden de $bd/2$, ya que debe mantener una lista de todos los nodos a profundidad i, y el número de nodos a profundidad $d/2$ es $bd/2$.

Como variante de este método, existe la **búsqueda dirigida por islas**, que es una generalización del BB en el que se establecen un conjunto de objetivos intermedios (**islas**), lo que suele reducir un problema de búsqueda grande en un conjunto de problemas de búsqueda menores. Si se van espaciando estas islas entre el nodo origen y el objetivo, se puede conseguir que su complejidad mejore, haciendo $mbd/m « bdmbd/m « bd$. Las cotas

decrecen, pero suele ser difícil garantizar la optimalidad, ya que habría que asegurar que todas las islas (u objetivos intermedios) están en el camino óptimo de la solución original.

El número de variantes que se han dado para mejorar la casuística de las búsquedas ciegas es enorme, y como nuestro objetivo es solo conocer las aproximaciones clásicas a la resolución de problemas, no entraremos en más detalles y pasaremos a estudiar otros tipos de búsquedas que resulten más eficientes en general.

4. El concepto de heurística. Como se construyen las heurísticas. Uso de las heurísticas en IA.

> En ciencias de la computación, dos objetivos fundamentales son encontrar algoritmos con buenos tiempos de ejecución y buenas soluciones, usualmente las óptimas. Una **heurística** es un algoritmo que abandona uno o ambos objetivos; por ejemplo, normalmente encuentran buenas soluciones, aunque no hay pruebas de que la solución no pueda ser arbitrariamente errónea en algunos casos; o se ejecuta razonablemente rápido, aunque no existe tampoco prueba de que siempre será así. Las heurísticas generalmente son usadas cuando no existe una solución óptima bajo las restricciones dadas (tiempo, espacio, etc.), o cuando no existe del todo.

A menudo, pueden encontrarse instancias concretas del problema donde la heurística producirá resultados muy malos o se ejecutará muy lentamente. Aun así, estas instancias concretas pueden ser ignoradas porque no deberían ocurrir nunca en la práctica por ser de origen teórico. Por tanto, el uso de heurísticas es muy común en el mundo real.

Ejemplo de Uso

Para ilustrar el funcionamiento de esta técnica, utilizaremos un ejemplo clásico de estructuras de datos e Inteligencia Artificial: el problema de las N reinas. La idea básica es colocar N reinas en un tablero de ajedrez de N×N casillas, de tal manera que ninguna de ellas pueda capturar a otra.
Este problema puede verse como uno de optimización en el que el objetivo es minimizar el número de colisiones (una colisión se presenta cuando una

reina puede capturar a otra). Si asumimos que cada reina se colocará siempre en una fila determinada de acuerdo a un cierto orden pre-establecido, podemos representar cualquier estado del tablero usando una permutación de la forma (p1, p2,..., pn), donde el valor de p1 representa la columna donde se colocará a la reina que está en la fila 1, y así sucesivamente.
Una de las operaciones básicas que el algoritmo utilizará es el intercambio de 2 elementos cualquiera de una permutación en la que haya colisiones. A fin de evitar la repetición de intercambios realizados recientemente, consideraremos tabú cualquier intercambio de este tipo. Debido a condiciones de simetría (intercambiar las reinas 1 y 4 es lo mismo que intercambiar las reinas 4 y 1), una estructura puede usarse para almacenar el número de iteraciones que faltan para permitir que 2 reinas intercambien posiciones nuevamente.

La restricción de los intercambios de 2 reinas puede ser ignorada si un cierto movimiento (prohibido) conduce a una mejor solución (este es el criterio de aspiración).

5. Los métodos de escalada. Caracterización general. Variantes.

Métodos de escalada

- **Se llaman de escalada (o de ascensión a la colina) porque tratan de elegir en cada paso un estado cuyo valor heurístico sea mayor que el del estado activo en ese momento.**
- **Se dividen en dos grupos:**
  - **Los métodos irrevocables**, que no preven la vuelta a un lugar del espacio de estados si el camino resulta inadecuado.
  - **Los métodos tentativos** en los que sí podemos volver hacia atrás si prevemos que el camino elegido no es el más adecuado.

Métodos de escalada irrevocables

- La lista ABIERTA no puede poseer más de un elemento, es decir, sólo se mantiene en expectativa un único camino.
- El modo en que se determina qué estado es el que sigue a uno dado, da lugar a dos variantes de los esquemas en escalada irrevocables:
  - **La escalada simple.** En este método, en el momento es que se encuentra un estado E que es más favorable que el que se está expandiendo, dicho estado E es devuelto sin generar el resto de estados hijos.
  - **La escalada por la máxima pendiente.** En este caso, se generan todos los hijos de un estado, se calculan sus valores heurísticos y se determina uno de los estados de mejor valor heurístico; se compara dicho valor con el de el estado expandido y si es mejor, se devuelve ese estado como expansión.

- En ambos casos, si ningún estado hijo es mejor que el estado que está siendo expandido, no se devuelve nada, lo que conllevará que la búsqueda termine sin haber encontrado un objetivo. Nótese que la escalada simple es mucho más dependiente que la escalada por la máxima pendiente del orden de disparo de las reglas (pese a que ésta última también lo es: el valor heurístico mejor puede ser alcanzado en varios hijos y, en ese caso, el método no dice nada sobre qué estado elegir).

Algoritmo de escalada simple

- **El algoritmo:**
  - 1. Denominar *m* al estado inicial y evaluarlo. Si es estado objetivo, entonces devolverlo y terminar, si no, convertirlo en estado actual.

    Asignar *m* a una varibla llamada *elegido.*
  - 2. Repetir hasta que se encuentre solución o hasta que una iteración completa no produzca un cambio en el estado actual:
    - 2.1 Expandir *m.* Para ello,
      1) Aplicar cualquier operador aplicable al estado actual *m* y obtener un nuevo estado Ei.
      2) Evaluar Ei:

         2.1 Si Ei es estado objetivo, devolverlo y terminar.

         2.1 Si Ei no es estado objetivo no es así, evaluar si Ei es mejor que el estado actual: (H(Ei) ->H*(elegido)*), en cuyo caso hacer *m*=Ei.
    - 2.2 Si *f(elegido) # f(m)* asignar *m=elegido, y volver a* ***2.***

Algoritmo de escalada por la máxima pendiente

- **El algoritmo:**
  - 1. Denominar m al estado inicial y evaluarlo. Si es estado objetivo, entonces devolverlo y terminar, si no, convertirlo en estado actual.

    Asignar *m* a una varibla llamada *elegido.*
  - 2. Repetir hasta que se encuentre solución o hasta que una iteración completa no produzca un cambio en el estado actual:
    - 2.1 Expandir *m* creando el **conjunto de todos sus sucesores.**
      A. Aplicar cada operador aplicable al estado actual *m* y conseguir nuevos estados E1, ... En.
      B. Evaluar E1 .... En. Si alguno es objetivo, devolverlo y terminar Si no es así, seleccionar el mejor H(Em)
      C. Si el mejor estado Em es mejor que el estado actual, hacer que Em sea el estado actual. Volver al paso **2.**
- **Frente a la escalada simple, considera todos los posibles estados nuevos (no el primero que es mejor, como hace la escalada simple)**

Inconvenientes de los métodos de escalada

- **El proceso de búsqueda puede no encontrar una solución cuando se encuentra con:**
  - **Máximo local**: un estado que es mejor que todos sus vecinos, pero no es mejor que otros estados de otros lugares. Puesto que todos los estados vecinos son peores el proceso se queda paralizado
  - **Meseta**: los estados vecinos tienen el mismo valor. El proceso, basado en comparación local, no discrimina la mejor dirección
- El origen de estos problemas es que el método de la escalada se basa en **comparaciones locales**, no explora exhaustivamente todas las consecuencias. Frente a otros métodos menos locales, tiene la ventaja de provocar una explosión combinatoria menor
- **Soluciones parciales:**
  - Almacenar la traza para poder ir hacia atrás, hasta un estado "prometedor" (o tan bueno como el que hemos dejado)
  - Dar un gran salto para seguir la búsqueda. Útil en el caso de mesetas

6. Características esenciales de los métodos "primero el mejor".

Idea de la búsqueda por primero el mejor

- Búsqueda por primero el mejor:
  - Analizar preferentemente los nodos con heurística más baja.
  - Ordenar la cola de abiertos por heurística, de menor a mayor
- También llamada búsqueda *voraz* o *codiciosa* (del inglés *"greedy"*)
  - Porque siempre elige expandir *lo que estima* que está más "cerca" del objetivo
- Su rendimiento dependerá de la bondad de la heurística usada.

Propiedades de la búsqueda por primero el mejor

- Complejidad en tiempo $O(r^m)$, donde:
  - r: factor de ramificación.
  - m: profundidad máxima del árbol de búsqueda.
- Complejidad en espacio: $O(r^m)$.
- En la práctica, el tiempo y espacio necesario depende del problema concreto y de la calidad de la heurística usada
- No es completa, en general.
  - Por ejemplo, una mala heurística podría hacer que se tomara un camino infinito.
- No es minimal (no garantiza soluciones con el menor número de operadores).
  - La heurística podría guiar hacia una solución no minimal

7. Elementos esenciales del algoritmo A*.

- **Usaremos dos listas de nodos (ABIERTA Y CERRADA)**
  - **Abierta**: nodos que se han generado y a los que se les ha aplicado la función heurística, pero que aún no han sido examinados (es decir, no se han generado sus sucesores)
    - Es decir, es una cola con prioridad en la que los elementos con mayor prioridad son aquellos que tienen un valor más prometedor de la función heurística.
  - **Cerrada**: nodos que ya se han examinado. Es necesaria para ver si cuando se genera un nuevo nodo ya ha sido generado con anterioridad.

- **Definiremos una función heurística f como la suma de dos funciones g y h:**
  - ***Función g***: es una medida del coste para ir desde el estado inicial hasta el nodo actual (suma de los costes o valores heurísticos de todos los nodos).
  - ***Función h:*** es una estimación del coste adicional necesario para alcanzar un nodo objetivo a partir del nodo actual, es decir, es una estimación de lo que me queda por recorrer hasta la meta.
  - La ***función combinada f*** una estimación del coste necesario para alcanzar un estado objetivo por el camino que se ha seguido para generar el nodo actual (si se puede generar por varios caminos el algoritmo se queda con el mejor).
  - **NOTA:** los nodos buenos deben poseer valores bajos.

- En cada paso se selecciona el nodo más prometedor que se haya generado hasta ese momento (función heurística).
- A continuación se expande el nodo elegido generando todos sus sucesores.
- Si alguna de ellos es meta el proceso acaba. Si no continúo con el algoritmo.
- Es parecido a la búsqueda en profundidad, pero si en una rama por la que estoy explorando no aparece la solución la rama puede parecer menos prometedora que otras por encima de ella y que se habían ignorado. Podemos pues abandonar la rama anterior y explorar la nueva.
- Sin embargo al vieja rama no se olvida. Su último nodo se almacena en el conjunto de nodos generados pero aún sin expandir.

1. Empezar con ABIERTA conteniendo sólo el nodo inicial. Poner el valor g de ese nodo a 0, su valor h al que corresponda, y su valor f a h+0, es decir, a h.

2. Inicializar CERRADA como una lista vacía.

3. Hasta que se encuentre una meta o de devuelva fallo realizar las siguientes acciones:

3.1 Si ABIERTA está vacía terminar con fallo; en caso contrario continuar.

3.2 Eliminar el nodo de ABIERTA que tenga un valor mínimo de *f*; llamar a este nodo *m* e introducirlo en la lista cerrada.

3.3 Si *m* es meta, abandonar el proceso iterativo iniciado en 2 devolviendo el camino recorrido (punteros a sus antepasados)

3.4 En caso contrario expandir *m* generando todos sus sucesores.

3.5 Para cada sucesor *n'* de *m*:

1) Crear un puntero de *n'* a *m*.

2) Calcular g(n')=g(m)+c(m,n'), tal que c(a,b) es el coste de pasar de *a* a *b*.

3) Si *n'* está en ABIERTA llamar *n* al nodo encontrado en dicha lista, añadirlo a los sucesores de *m* y realizar el siguiente paso:

3.1) Si g(n')<g(n), entonces redirigir el puntero de *n* a *m* y cambiar el camino de menor coste encontrado a n desde la raíz; *g(n)=g(n') y f(n)=g(n')+h(n).*

4) Si *n'* no cumple 3), comprobar si está en cerrada; llamar *n* al nodo encontrado en dicha lista y realizar las siguientes acciones:

Si 3.1) no se cumple, abandonar 4); en caso contrario propagar el nuevo menor coste *g(n')* (por lo que también actualizarán los valores de *f* correspondientes(que llamaremos n_i tal que i=1,2,…, siendo sus costes anteriores g(n_i)), realizando un *recorrido en profundidad* de éstos, empezando en *n'* y teniendo en cuenta las siguientes consideraciones:

4.1) Para los nodos descendientes n_i cuyo puntero (que debe apuntar siempre al mejor predecesor hasta ese momento) conduzca hacia el nodo n_i , actualizar $g(n_i)$=g(n_i') y f(n_i)=g(n_i‘)+h(n_i) y seguir el recorrido hasta que se encuentre un n_i que no tenga más sucesores calculados o se llegue a un nodo en que ya ocurra que $g(n_i)$=$g(n_i$‘), en cuyo caso se habría producido un ciclo y también habría que terminar la propagación.

4.2) Para los nodos descendientes n_i cuyo puntero no conduzca hacia el nodo *n'*, comprobar si $g(n_i\text{ '}) < g(n_i)$, en cuyo caso se debe actualizar el puntero para que conduzca hacia el nodo n' (mejor camino desde la raíz encontrado hasta ese momento) y se continúa el proceso de propagación.

5) Si n´no está en ABIERTA o en CERRADA, calcular *h(n')* y *f(n')=g(n')+h(n')*, introducirlo en ABIERTA y añadirlo a la lista de sucesores de *m*.

- **Relaciones con otros métodos**
  - Si para todo *n* y para todo *m h(n)=0* y *c(m,n)*=1, es decir, el coste para pasar de m a n es 1 entonces A* se convierte en un proceso de búsqueda en amplitud.
- **Características**
  - Es un método completo de búsqueda, es decir, termina encontrando la solución cuando ésta exista para cualquier tipo de grafos.
  - Es **admisible**. Es decir, no sólo encuentra la solución sino que la que encuentra es la optima si se cumple la siguiente condición:
    - **Para todo *n* h(n) es menor o igual que h*(n), es decir, si la función heurística que estima la distancia a la meta nunca puede superar la distancia real existente entonces A* garantiza encontrar la solución óptima.**

8. Elementos esenciales de un algoritmo genético.

La aplicación más común de los algoritmos genéticos ha sido la solución de problemas de optimización, en donde han mostrado ser muy eficientes y confiables. Sin embargo, no todos los problemas pudieran ser apropiados para la técnica, y se recomienda en general tomar en cuenta las siguientes características del mismo antes de intentar usarla:

• Su espacio de búsqueda (i.e., sus posibles soluciones) debe estar delimitado dentro de un cierto rango.

• Debe poderse definir una función de aptitud que nos indique qué tan buena o mala es una cierta respuesta.

• Las soluciones deben codificarse de una forma que resulte relativamente fácil de implementar en la computadora.

El primer punto es muy importante, y lo más recomendable es intentar resolver problemas que tengan espacios de búsqueda discretos aunque éstos sean muy grandes. Sin embargo, también podrá intentarse usar la técnica con espacios de búsqueda continuos, pero preferentemente cuando exista un rango de soluciones relativamente pequeño.

La función de aptitud no es más que la función objetivo de nuestro problema de optimización. El algoritmo genético únicamente maximiza, pero la minimización puede realizarse fácilmente utilizando el recíproco de la función maximizante (debe cuidarse, por supuesto, que el recíproco de la función no genere una división por cero). Una característica que debe tener esta función es que tiene ser capaz de "castigar" a las malas soluciones, y de "premiar" a las buenas, de forma que sean estas últimas las que se propaguen con mayor rapidez.

La codificación más común de las soluciones es a través de cadenas binarias, aunque se han utilizado también números reales y letras. El primero de estos esquemas ha gozado de mucha popularidad debido a que es el que propuso originalmente Holland, y además porque resulta muy sencillo de implementar.