# Búsqueda / Search

Los problemas de búsqueda involucran a un agente al que se le asigna un estado inicial y un estado objetivo, y devuelve una solución para llegar del primero al segundo. Una aplicación de navegación utiliza un proceso de búsqueda, donde el agente (la parte pensante del programa) recibe como entrada su ubicación actual y su destino deseado y, en función de un algoritmo de búsqueda, devuelve una ruta sugerida.

Sin embargo, existen otros problemas de búsqueda, como rompecabezas o laberintos.



Para encontrar una solución a un rompecabezas de 15, se necesitaría usar un algoritmo de búsqueda.

• Agente

Entidad que percibe su entorno y actúa en función de él. En una aplicación de navegación, por ejemplo, el agente sería una representación de un automóvil que debe decidir qué acciones realizar para llegar a su destino.

• Estado

Configuración de un agente en su entorno. Por ejemplo, en un rompecabezas de 15, un estado es cualquier forma en que se ordenan todos los números en el tablero.

o Estado inicial

El estado desde el que comienza el algoritmo de búsqueda. En una aplicación de navegación, sería la ubicación actual.

• Acciones

Opciones que se pueden realizar en un estado. Más precisamente, las acciones se pueden definir como una función. Al recibir el estado ***s*** como entrada, ***Action(s)*** devuelve como salida el conjunto de acciones que se pueden ejecutar en el estado ***s***.

Por ejemplo, en un rompecabezas de 15, las acciones de un estado dado son las formas en que se pueden deslizar los cuadrados en la configuración actual (4 si el cuadrado vacío está en el medio, 3 si está al lado de un lado, 2 si está en la esquina).

• Modelo de transición

Una descripción de qué estado resulta de realizar cualquier acción aplicable en cualquier estado. Más precisamente, el modelo de transición se puede definir como una función. Al recibir el estado ***s*** y la acción ***a*** como entrada, ***Results(s, a)*** devuelve el estado resultante de realizar la acción ***a*** en el estado ***s***.

Por ejemplo, dada una cierta configuración de un rompecabezas de 15 (estado s), mover un cuadrado en cualquier dirección (acción a) traerá una nueva configuración del rompecabezas (el nuevo estado).

• Espacio de estados

El conjunto de todos los estados alcanzables desde el estado inicial mediante cualquier secuencia de acciones.

Por ejemplo, en un rompecabezas de 15, el espacio de estados consta de todas las 16!/2 configuraciones en el tablero a las que se puede llegar desde cualquier estado inicial. El espacio de estados se puede visualizar como un gráfico dirigido con estados, representados como nodos, y acciones, representadas como flechas entre nodos.

Prueba de objetivo / Goal Test

La condición que determina si un estado dado es un estado objetivo.

Por ejemplo, en una aplicación de navegación, la prueba de objetivo sería si la ubicación actual del agente (la representación del automóvil) está en el destino. Si es así, problema resuelto. Si no es así, continuamos la búsqueda.

• Costo de ruta

Un costo numérico asociado con una ruta dada.

Por ejemplo, una aplicación de navegación no solo lo lleva a su objetivo; lo hace mientras minimiza el costo de la ruta, encontrando la forma más rápida posible para que usted llegue a su estado objetivo.

Solución de problemas de búsqueda

* Solución

Una secuencia de acciones que lleva del estado inicial al estado objetivo.

* Solución óptima

Una solución que tiene el menor costo de ruta entre todas las soluciones.

En un proceso de búsqueda, los datos a menudo se almacenan en un nodo, una estructura de datos que contiene los siguientes datos:

• Un estado

• Su nodo padre, a través del cual se generó el nodo actual

• La acción que se aplicó al estado del padre para llegar al nodo actual

• El costo de la ruta desde el estado inicial hasta este nodo

Los nodos contienen información que los hace muy útiles para los fines de los algoritmos de búsqueda. Contienen un estado, que se puede verificar utilizando la prueba de objetivo para ver si es el estado final. Si lo es, el costo de ruta del nodo se puede comparar con los costos de ruta de otros nodos, lo que permite elegir la solución óptima.

Una vez que se elige el nodo, en virtud de almacenar el nodo padre y la acción que condujo desde el padre al nodo actual, es posible rastrear cada paso del camino desde el estado inicial hasta este nodo, y esta secuencia de acciones es la solución.

Sin embargo, los nodos son simplemente una estructura de datos: no buscan, guardan información. Para buscar realmente, utilizamos la frontera, el mecanismo que "administra" los nodos. La frontera comienza conteniendo un estado inicial y un conjunto vacío de elementos explorados, y luego repite las siguientes acciones hasta que se llega a una solución:

Repetir:

1. Si la frontera está vacía,

* Detenerse. No hay solución para el problema.

2. Eliminar un nodo de la frontera. Este es el nodo que se considerará.

3. Si el nodo contiene el estado objetivo,

* Devolver la solución. Detenerse.

De lo contrario / ELSE,

\* Expandir el nodo (encontrar todos los nodos nuevos a los que se podría llegar desde este nodo) y agregar los nodos resultantes a la frontera.

\*Agrega el nodo actual al conjunto explorado.

# Búsqueda en profundidad

En la descripción anterior de la frontera, no se mencionó un aspecto. En la etapa 2 del pseudocódigo anterior, ¿qué nodo se debe eliminar? Esta elección tiene implicaciones en la calidad de la solución y en la rapidez con la que se logra.

Hay varias formas de abordar la cuestión de qué nodos se deben considerar primero, dos de las cuales se pueden representar mediante las estructuras de datos de pila/ queue (en la búsqueda en profundidad) y cola/stack .

Comenzamos con el enfoque de búsqueda en profundidad (DFS).

Un algoritmo de búsqueda en profundidad agota cada una de las direcciones antes de intentar otra dirección. En estos casos, la frontera se administra como una estructura de datos de pila. El lema que debe recordar aquí es "último en entrar, primero en salir". Después de que se agregan nodos a la frontera, el primer nodo que se elimina y se considera es el último que se agrega. Esto da como resultado un algoritmo de búsqueda que profundiza lo más posible en la primera dirección que se interpone en su camino y deja todas las demás direcciones para más adelante.

(Ejemplo: tomemos una situación en la que estamos buscando nuestras llaves. En un enfoque de búsqueda en profundidad, si decidimos empezar a buscar en nuestros pantalones, primero revisaríamos cada uno de los bolsillos, vaciando cada uno de ellos y revisando el contenido cuidadosamente. Dejaremos de buscar en nuestros pantalones y comenzaremos a buscar en otro lugar solo cuando hayamos agotado por completo la búsqueda en cada uno de los bolsillos de nuestros pantalones).

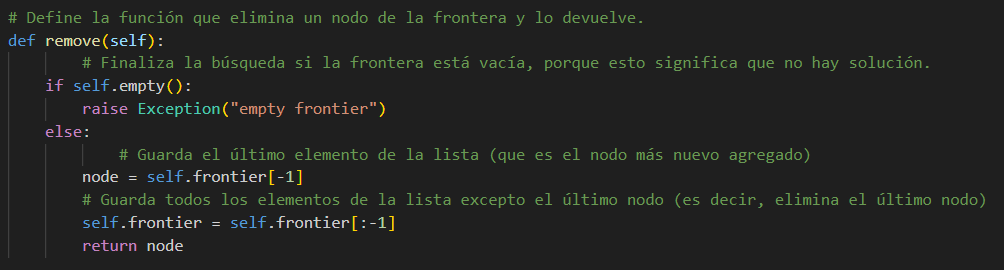
• Ventajas:

* En el mejor de los casos, este algoritmo es el más rápido. Si “tiene suerte” y siempre elige el camino correcto hacia la solución (por casualidad), entonces la búsqueda en profundidad tarda el menor tiempo posible en llegar a una solución.

• Desventajas:

* Es posible que la solución encontrada no sea la óptima.
* En el peor de los casos, este algoritmo explorará todos los caminos posibles antes de encontrar la solución, por lo que tardará el mayor tiempo posible en llegar a ella.

Code example:



# Búsqueda en amplitud

El opuesto de la búsqueda en profundidad sería la búsqueda en amplitud (BFS).

Un algoritmo de búsqueda en amplitud seguirá múltiples direcciones al mismo tiempo, dando un paso en cada dirección posible antes de dar el segundo paso en cada dirección.

En este caso, la frontera se administra como una estructura de datos de cola. El lema que debe recordar aquí es "primero en entrar, primero en salir". En este caso, todos los nodos nuevos se suman en línea y los nodos se consideran en función de cuál se agregó primero (¡el primero en llegar es el primero en ser atendido!).

Esto da como resultado un algoritmo de búsqueda que da un paso en cada dirección posible antes de dar un segundo paso en cualquier dirección. (Ejemplo: supongamos que estás en una situación en la que estás buscando tus llaves. En este caso, si empiezas por tus pantalones, buscarás en tu bolsillo derecho. Después, en lugar de buscar en tu bolsillo izquierdo, buscarás en un cajón. Después en la mesa. Y así sucesivamente, en todos los lugares que se te ocurran. Solo después de haber agotado todos los lugares volverás a tus pantalones y buscarás en el siguiente bolsillo).

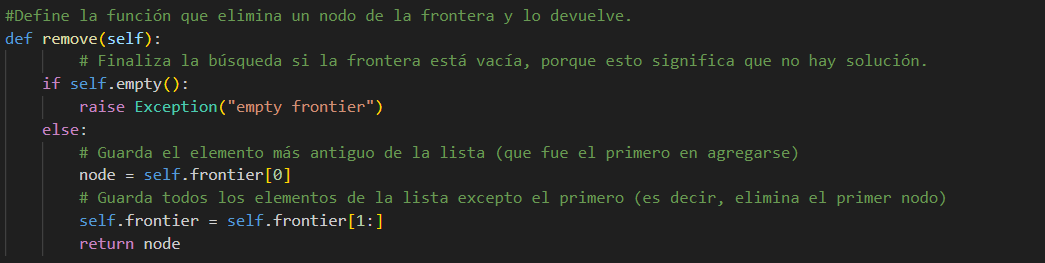
• Pros:

* Se garantiza que este algoritmo encontrará la solución óptima.

• Contras:

* Es casi seguro que este algoritmo tardará más que el tiempo mínimo en ejecutarse.
* En el peor de los casos, este algoritmo tarda el mayor tiempo posible en ejecutarse.

Code example:



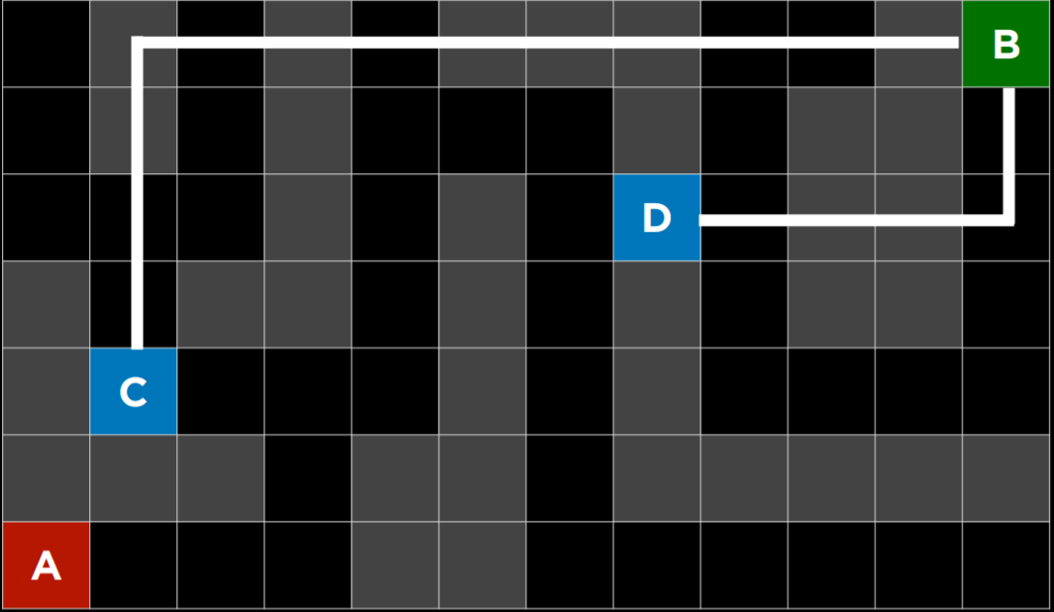
# Greedy Best-First Search

Tanto los algoritmos de búsqueda en amplitud como en profundidad son algoritmos de búsqueda no informados. Es decir, estos algoritmos no utilizan ningún conocimiento sobre el problema que no hayan adquirido a través de su propia exploración.

Sin embargo, lo más frecuente es que, de hecho, se disponga de algún conocimiento sobre el problema. Por ejemplo, cuando un humano resuelve un laberinto entra en una intersección, puede ver qué camino va en la dirección general de la solución y cuál no. La IA puede hacer lo mismo. Un tipo de algoritmo que considera el conocimiento adicional para intentar mejorar su rendimiento se denomina algoritmo de búsqueda informado.

E Greedy Best-First Search expande el nodo que está más cerca del objetivo, según lo determinado por una función ***heurística h(n)***.

Como sugiere su nombre, la función estima qué tan cerca del objetivo está el siguiente nodo, pero puede estar equivocada. La eficiencia del Greedy Best-First Search depende de qué tan buena sea la función heurística. Por ejemplo, en un laberinto, un algoritmo puede utilizar una función heurística que se basa en la distancia de Manhattan entre los posibles nodos y el final del laberinto. La distancia Manhattan ignora las paredes y cuenta cuántos pasos hacia arriba, hacia abajo o hacia los lados se necesitarían para llegar desde una ubicación a la ubicación objetivo. Esta es una estimación fácil que se puede derivar en función de las coordenadas (x, y) de la ubicación actual y la ubicación objetivo.



Distancia de Manhattan

Sin embargo, es importante destacar que, como sucede con cualquier heurística, puede salir mal y hacer que el algoritmo tome un camino más lento del que hubiera tomado de otro modo. Es posible que un algoritmo de búsqueda desinformado proporcione una mejor solución más rápido, pero es menos probable que lo haga que un algoritmo informado.

[**A\* Search**](https://cs50.harvard.edu/ai/2024/notes/0/#a-search)

La búsqueda A\*, un desarrollo del algoritmo greedy best-first, considera no solo h(n), el costo estimado desde la ubicación actual hasta el objetivo, sino también g(n), el costo acumulado hasta la ubicación actual.

Al combinar ambos valores, el algoritmo tiene una forma más precisa de determinar el costo de la solución y optimizar sus opciones sobre la marcha. El algoritmo lleva un registro de (costo del camino hasta ahora + costo estimado hasta el objetivo) y, una vez que excede el costo estimado de alguna opción anterior, el algoritmo descartará el camino actual y volverá a la opción anterior, evitando así seguir un camino largo e ineficiente que h(n) marcó erróneamente como el mejor.

Una vez más, dado que este algoritmo también se basa en una heurística, es tan bueno como la heurística que emplea. Es posible que en algunas situaciones sea menos eficiente que la búsqueda voraz de mejor primero o incluso los algoritmos desinformados. Para que la búsqueda A\* sea óptima, la función heurística, h(n), debe ser:

1. Admisible, o nunca sobreestimar el costo real, y

2. Consistente, lo que significa que el costo estimado de la ruta hacia el objetivo de un nuevo nodo además del costo de la transición hacia él desde el nodo anterior es mayor o igual que el costo estimado de la ruta hacia el objetivo del nodo anterior. Para ponerlo en forma de ecuación, h(n) es consistente si para cada nodo n y nodo sucesor n’ con costo de paso c, h(n) ≤ h(n’) + c.

# [**Adversarial Search**](https://cs50.harvard.edu/ai/2024/notes/0/#adversarial-search)

Mientras que, anteriormente, hemos discutido algoritmos que necesitan encontrar una respuesta a una pregunta, en la búsqueda adversarial el algoritmo se enfrenta a un oponente que intenta lograr el objetivo opuesto.

A menudo, la IA que utiliza la búsqueda adversarial se encuentra en juegos, como el gato.

## Minimax

Un tipo de algoritmo en la búsqueda adversarial, Minimax representa las condiciones ganadoras como (-1) para un lado y (+1) para el otro lado. Las acciones posteriores serán impulsadas por estas condiciones, con el lado que minimiza tratando de obtener la puntuación más baja, y el lado que maximiza tratando de obtener la puntuación más alta.

Representación de una IA del juego de gato:

• S₀: Estado inicial (en nuestro caso, un tablero vacío de 3X3)

• Jugador(es): una función que, dado un estado s, devuelve a qué jugador le toca (X u O).

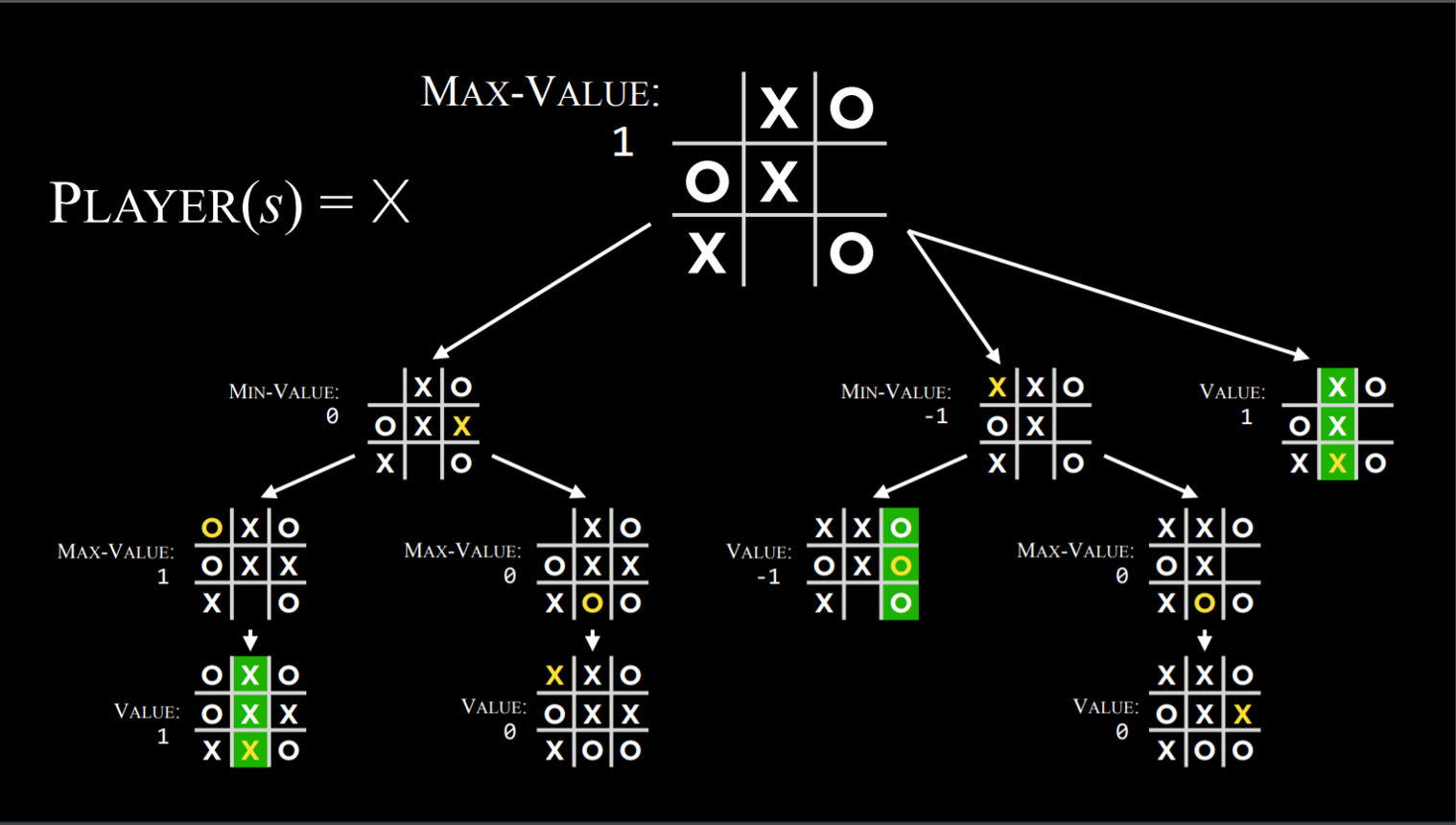
• Acciones: una función que, dado un estado s, devuelve todos los movimientos legales en este estado (qué lugares están libres en el tablero).

• Result(s, a): función que, dado un estado s y una acción a, devuelve un nuevo estado. Este es el tablero que resultó de realizar la acción a en el estado s (hacer un movimiento en el juego).

• Terminal(es): función que, dado un estado s, verifica si este es el último paso del juego, es decir, si alguien ganó o si hay un empate. Devuelve True si el juego ha terminado, False en caso contrario.

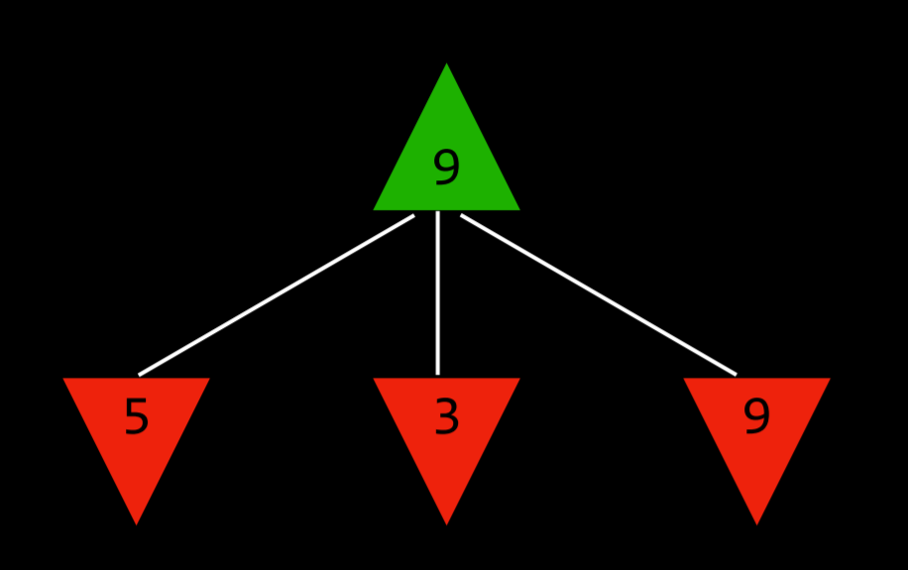
• Utility(s): función que, dado un estado terminal s, devuelve el valor de utilidad del estado: -1, 0 o 1.

Cómo funciona el algoritmo:

De forma recursiva, el algoritmo simula todos los juegos posibles que pueden tener lugar comenzando en el estado actual y hasta que se alcanza un estado terminal. Cada estado terminal se valora como (-1), 0 o (+1).

Algoritmo Minimax en el juego del gato

Al saber en función del estado a quien le toca jugar, el algoritmo puede saber si el jugador actual, al jugar óptimamente, elegirá la acción que lleve a un estado con un valor más bajo o más alto. De esta manera, alternando entre minimizar y maximizar, el algoritmo crea valores para el estado que resultaría de cada posible acción. Para dar un ejemplo más concreto, podemos imaginar que el jugador que maximiza pregunta en cada turno: “si realizo esta acción, resultará un nuevo estado. Si el jugador que minimiza juega óptimamente, ¿qué acción puede realizar ese jugador para llegar al valor más bajo?”. Sin embargo, para responder a esta pregunta, el jugador que maximiza tiene que preguntarse: “para saber qué hará el jugador que minimiza, necesito simular el mismo proceso en la mente del minimizador: el jugador que minimiza intentará preguntar: ‘si realizo esta acción, ¿qué acción puede realizar el jugador que maximiza para llegar al valor más alto?’”. Este es un proceso recursivo, y puede resultar difícil de entender;

Observar el pseudocódigo que aparece a continuación puede ser de ayuda. Finalmente, a través de este proceso de razonamiento recursivo, el jugador que maximiza genera valores para cada estado que podrían resultar de todas las acciones posibles en el estado actual. Después de tener estos valores, el jugador que maximiza elige el más alto.

El maximizador considera los posibles valores de los estados futuros.

Para ponerlo en pseudocódigo, el algoritmo Minimax funciona de la siguiente manera:

• Dado un estado s

* El jugador que maximiza elige la acción a en Acciones(s) que produce el valor más alto de Valor-Mín(Resultado(s, a)).
* El jugador que minimiza elige la acción a en Acciones(s) que produce el valor más bajo de Valor-Máximo(Resultado(s, a)).

• Función Valor-Máximo(estado)

* v = -∞
* si Terminal(estado):

devuelve Utilidad(estado)

* para la acción en Acciones(estado):

v = Máx(v, Valor-Mín(Resultado(estado, acción)))

devuelve v

• Función Valor-Mín(estado):

* v = ∞
* si Terminal(estado):

devuelve Utilidad(estado)

* para la acción en Acciones(estado):

v = Mín(v, Valor-Máximo(Resultado(estado, acción)))

devuelve v

Poda Alfa-Beta

Una forma de optimizar Minimax, la Poda Alfa-Beta omite algunos de los cálculos recursivos que son decididamente desfavorables. Después de establecer el valor de una acción, si hay evidencia inicial de que la siguiente acción puede llevar al oponente a obtener una puntuación mejor que la acción ya establecida, no hay necesidad de investigar más sobre esta acción porque será decididamente menos favorable que la establecida previamente.

Esto se muestra más fácilmente con un ejemplo: un jugador que maximiza sabe que, en el siguiente paso, el jugador que minimiza intentará lograr la puntuación más baja. Supongamos que el jugador que maximiza tiene tres acciones posibles y la primera tiene un valor de 4. Entonces el jugador comienza a generar el valor para la siguiente acción. Para ello, el jugador genera los valores de las acciones del minimizador si el jugador actual realiza esta acción, sabiendo que el minimizador elegirá la más baja. Sin embargo, antes de terminar el cálculo para todas las acciones posibles del minimizador, el jugador ve que una de las opciones tiene un valor de tres. Esto significa que no hay razón para seguir explorando las otras posibles acciones del jugador que minimiza. El valor de la acción aún no valorada no importa, ya sea 10 o (-10). Si el valor es 10, el minimizador elegirá la opción más baja, 3, que ya es peor que la 4 preestablecida. Si la acción aún no valorada resultara ser (-10), el minimizador elegirá esta opción, (-10), que es aún más desfavorable para el maximizador. Por lo tanto, calcular posibles acciones adicionales para el minimizador en este punto es irrelevante para el maximizador, porque el jugador que maximiza ya tiene una opción inequívocamente mejor cuyo valor es 4.

