Búsqueda en IA (parte 1)

Jorge A. Baier

Departamento de Ciencia de la Computación
Pontificia Universidad Católica de Chile

Santiago, Chile

Búsqueda

- Todo problema en el que es necesario *encontrar* una solución es un problema de búsqueda.
- Un algoritmo se dice *de búsqueda* se mueve a través de un espacio de búsqueda para encontrar una solución.
- Se usa un algoritmo de búsqueda en problemas en donde *no se tiene* una solución algorítmica.
- Posibles ejemplos: planificar un viaje, jugar ajedrez, resolver un puzle.

Ejemplos

Un solo agente:

- Cubo Rubik, Puzle de $(n^2 - 1)$.
- Sudoku, Atomix
- Navegación de Robots, Planificación de Movimientos
- Razonamiento Hipotético
- Verificación de Software

Múltiples agentes:

- Damas, Ajedrez, Go, ...
- Bridge, Poker, ...
- Backgammon

Casos de Éxito

- El espacio de búsqueda del cubo Rubik tiene

$$43{,}252{,}003{,}274{,}489{,}856{,}000 \approx 4 \cdot 10^{19}$$

  estados. Sin embargo, un *solver* para este problema "sólo" necesitó explorar 10^{12} estados para encontrar una solución óptima de 18 pasos en 17 días. (El problema más difícil posible tiene 20 movidas)
- El juegos de las damas tiene un espacio de estados de 10^{20} y el ajedrez 10^{44}. Para ambos juegos, existen programas que buscan mejor que cualquier humano. Las damas, de hecho, está *resuelto*.
- Búsqueda es usado en aplicaciones industriales: planificación de brazos industriales, *debugging*, diagnóstico de circuitos eléctricos, etc.

Mundos Determinísticos, con Un Agente

- Un espacio de estados $\mathcal{S}$.
- Un conjunto $\mathcal{A}$ de operadores. Un operador $a \in \mathcal{A}$ es una función *parcial*
$$a : \mathcal{S} \mapsto \mathcal{S}.$$
- Por cada estado, un conjunto $A(s) \subseteq \mathcal{A}$ de *operadores aplicables* en s. Si $a \in A(s)$, entonces $a(s)$ está definida. Definimos
$$Succ(s) = \{a(s) \mid a \in A(s)\}$$
- Una función de costo $c : \mathcal{A} \to \mathbb{R}^+$.
- Un estado inicial s_0.
- Un conjunto de estados finales G.

Solución a un Problema de Búsqueda

- Una secuencia de operadores $o_0 o_1 \ldots o_n$ es aplicable en s_0 ssi $s_{i+1} = o_i(s_i)$ está definido, para todo $i \in \{0, \ldots, n\}$.

- Una secuencia aplicable de operadores $o_0 o_1 \ldots o_n$ es una solución al problema ssi cuando $s_{i+1} = o_i(s_i)$, para todo $i \in \{0, \ldots, n\}$, $s_{n+1} \in G$.

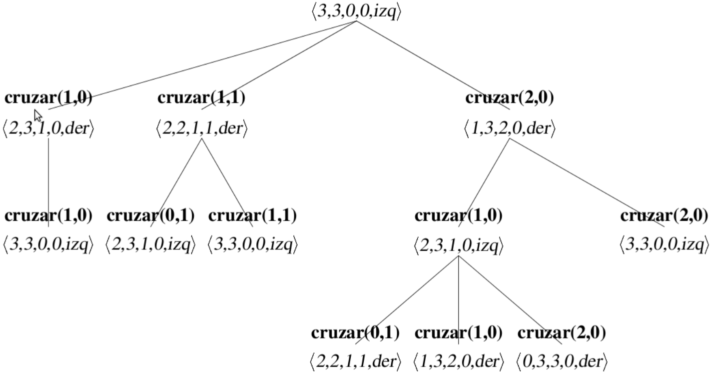
Otro Ejemplo: Misioneros y Caníbales

En este problema hay tres caníbales, tres misioneros, un río y un bote. Los caníbales, los misioneros y el bote se encuentran en una rivera del río. Los seis sujetos deben cruzar el río, pero el bote sólo permite trasladar a dos personas a la vez. Se debe encontrar una secuencia de movimientos de personas en el bote que permita cruzar a los seis individuos de manera segura. No se debe permitir que hayan más caníbales que misioneros en algún lado del río algún momento.

Ejercicio: Formalice este problema como un problema de búsqueda.

Espacio de Búsqueda para Misioneros y Caníbales

Una vista parcial del espacio de búsqueda.



Búsqueda Genérica

El siguiente es un algoritmo de búsqueda genérico.

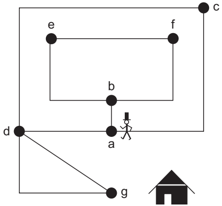
Input: Un problema de búsqueda $(\mathbf{S}, A, s_0, G)$

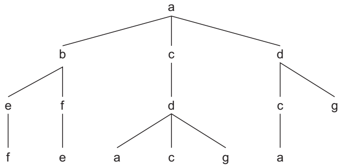
Output: Un nodo objetivo

$Closed \leftarrow \emptyset$
$Open \leftarrow \{s_0\}$
$s_0.parent = null$
while $Open \neq \emptyset$
    $u \leftarrow \texttt{Extraer}(Open)$
    Inserta u en $Closed$
    for each $v \in Succ(u) \setminus (Open \cup Closed)$
        $v.parent = u$
        if $v \in G$ **return** v
    $\texttt{Mezcla}(Succ(u), Open)$

Ejemplo (Heuristic Search; Edelkamp, Schrödl, 2011)

Consideremos el grafo de la izquierda y el árbol del espacio de búsqueda de la derecha.





Búsqueda en Profundidad (*Depth-First Search*)

- Usualmente abreviado como DFS.
- Resulta de implementar a *Open* como un stack.
- Siempre se extrae el elemento al tope de *Open* (línea 4; alg. principal).
- La función `Mezcla` agrega los elementos de $Succ(u)$ que no están en $Open \cup Closed$ al tope del stack.

Ejemplo: En pizarra.

Búsqueda en Amplitud (*Breadth-First Search*)

- Resulta de implementar a *Open* como una cola.
- Siempre se extrae el primer elemento al principio de *Open* (línea 4; alg. principal).
- La función `Mezcla` agrega los elementos de $Succ(u)$ que no están en $Open \cup Closed$ al final de la cola.

Ejemplo: En pizarra.

Propiedades

Teorema

Si el espacio de estados es finito, búsqueda en profundidad con detección de ciclos es completo (es decir, encuentra una solución si ésta existe).

Propiedades

Teorema

Si el espacio de estados es finito, búsqueda en profundidad con detección de ciclos es completo (es decir, encuentra una solución si ésta existe).

Teorema

Si el espacio de búsqueda es finito, búsqueda en amplitud es completo y óptimo para problemas de búsqueda con costos uniformes.

Tiempo y Espacio

Para los siguientes resultados, suponemos:

- b: factor de ramificación promedio.
- p: profundidad a la que se encuentra la solución.
- m: largo de la rama más larga del árbol de búsqueda.

Teorema

La memoria usada por DFS es $\mathcal{O}(bm)$, mientras que breadth-first necesita memoria de tamaño $\mathcal{O}(b^p)$.

Teorema

DFS requiere tiempo $\mathcal{O}(b^m)$, mientras que breadth-first necesita tiempo $\mathcal{O}(b^p)$.

Lo mejor de los dos mundos

Profundidad Limitada

Funciona como DFS, pero recibe como parámetro un límite ℓ de profundidad para la búsqueda. Se ejecuta DFS sobre el subárbol de profundidad ℓ del espacio de búsqueda.

Lo mejor de los dos mundos

Profundidad Limitada
Funciona como DFS, pero recibe como parámetro un límite ℓ de profundidad para la búsqueda. Se ejecuta DFS sobre el subárbol de profundidad ℓ del espacio de búsqueda.

Profundización Iterativa (*Iterative Deepening DFS*)

1. ℓ=1;
2. realice búsqueda en profundidad limitada con límite ℓ.
3. si hubo éxito, retorne el estado encontrado; en otro caso incremente ℓ y vuelva al paso anterior.

Resultados sobre IDDFS

Teorema

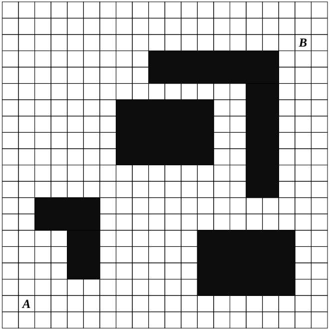
Profundización Iterativa es completo.

- b: factor de ramificación promedio.
- p: profundidad a la que se encuentra la solución.
- m: largo de la rama más larga del árbol de búsqueda.

Teorema

El tiempo requerido por IDDFS es $\mathcal{O}(b^p)$ y memoria de tamaño $\mathcal{O}(bp)$.

¿Qué podemos hacer para mejorar la búsqueda en estos casos?



Problema: ir de A a B

1	2	3
	5	4
6	7	8

$\longrightarrow$

1	2	3
8		4
7	6	5

Búsqueda el Mejor Primero (*Best-First Search*)

El algoritmo *el mejor primero*, intuitivamente:

- Mantiene una lista de *Open* y *Closed*.
- Funciona como DFS, pero:
- Los nodos en *Open* tienen asociados una calidad.
- Siempre extrae de *Open* el nodo de mejor calidad.
- Un estado sucesor es descartado si está en *Closed* con mejor o igual calidad.

Función Heurística

- En búsqueda informada, usamos una función de estimación del costo de un nodo del árbol de búsqueda a una solución. La denotamos como
$$h(n)$$
- En el problema de navegación, si
$$\Delta x = x_{obj} - x, \quad \Delta y = y_{obj} - y,$$
donde (x, y) es la posición actual y (x_{obj}, y_{obj}) es el objetivo. La siguiente es una posible heurística:
$$h(x, y) = |\Delta x - \Delta y| + \sqrt{2}\min\{|\Delta x|, |\Delta y|\}$$
- ¿Qué pasa si ordenamos *Open* usando $h(n)$? (Ejemplo en Pizarra)

Incorporando el Costo

- Como vimos en el ejemplo, usar sólo h conduce a soluciones no óptimas.
- Es posible encontrar soluciones óptimas al incorporar el *costo* incurrido hasta llegar a un nodo n.
- Denotamos este costo como $g(n)$.
- Luego, podemos ordenar la frontera de búsqueda por la siguiente función:

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

Algoritmo Principal

Algoritmo A*

Input: Un problema de búsqueda $(\mathbf{S}, A, s_0, G)$

Output: Un nodo objetivo

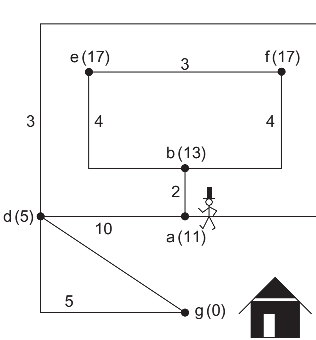
for each $s \in \mathcal{S}$ **do** $g(s) \leftarrow \infty$
$Open \leftarrow \{s_0\}$
$g(s_0) \leftarrow 0;\ f(s_0) \leftarrow h(s_0)$
while $Open \neq \emptyset$
    Extrae un u desde $Open$ con menor valor-f
    if u es objetivo **return** u
    for each $v \in Succ(u)$ **do**
        Insertar v

El Procedimiento *Insertar*

Insertar v **en Open**

$cost_v = g(u) + c(u, v)$ // el costo de llegar a v por u
if $cost_v \geq g(v)$ **return** // seguimos solo si $cost_v < g(v)$
$parent(v) \leftarrow u$
$g(v) \leftarrow cost_v$
$f(v) \leftarrow g(v) + h(v)$
if $v \in Open$ **then** Reordenar *Open* // depende de la impl.
else Insertar v en *Open*

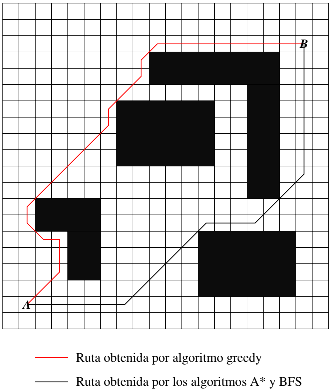
Un ejemplo



Entre paréntesis, $h(n)$.

A* y Greedy

- Si usamos $f(n) = h(n)$ en A*, entonces el algoritmo resultante es *greedy best-first search* (ambicioso).
- Los algoritmos ambiciosos encuentran soluciones más rápidamente, sacrificando la calidad de la solución.



Optimalidad de A*

Partiremos con algunas definiciones

Definición

Para un estado s, denotamos por $h^*(s)$ al costo de un camino óptimo desde s a un estado objetivo.

Definición (Admisibilidad)

Una función heurística h se dice *admisible*, si para todo s:

$$h(s) \leq h^*(s)$$

Teorema (Optimalidad de A*)

Si h es admisible, entonces A*, usado con h, encuentra una solución óptima si esta existe.

Demostración: Pizarra.

Heurísticas Consistentes

Definición (Heurísticas Consistentes)

Una heurística se dice *consistente* ssi

- $h(s) = 0$, para todo $s \in G$.
- $h(s) \leq c(s, s') + h(s')$, para todo vecino s' de s.

Teorema

Si h es consistente, entonces h es admisible.

Teorema

Cuando A* es usado con una heurística admisible, cuando A* expande un nodo v, $g(v)$ contiene el costo del camino óptimo desde s_0 a v.

El anterior teorema tiene un potencial impacto en la forma de implementar A*.

La mayor es la mejor

Teorema

Si h_1 y h_2 son consistentes y $h_1 \geq h_2$, entonces A*, usado con h_2, expande todos los nodos que A* expande cuando es usado con h_1.

Como conclusión tenemos que h_1 es "mejor" que h_2 en la práctica.

Encontrando Heurísticas Admisibles

- Una estrategia simple: *relajar* el problema.
- La heurística es el costo de resolver el problema relajado.
- Ejemplo:

2		3
1	8	6
7	5	4

Estado Inicial

1	2	3
8		4
7	6	5

Objetivo

- Los operadores respetan las siguientes restricciones:
  1. Un azulejo sólo se puede mover a un cuadrado vecino.
  2. Un azulejo sólo se puede mover a un cuadrado desocupado.

Heurísticas en Nuestro Ejemplo

- Si relajamos ambas restricciones:

$$h_1 = \text{“número de azulejos en la posición incorrecta”}$$

- Si relajamos la restricción 2:

$$h_2 = \text{“suma de la distancia } \textit{manhattan} \text{ de cada azulejo”}$$

¿cuál es mejor?

Heurísticas en Nuestro Ejemplo

- Si relajamos ambas restricciones:

$$h_1 = \text{“número de azulejos en la posición incorrecta”}$$

- Si relajamos la restricción 2:

$$h_2 = \text{“suma de la distancia } \textit{manhattan} \text{ de cada azulejo”}$$

¿cuál es mejor?

	Search Cost		
d	IDS	A*(h_1)	A*(h_2)
2	10	6	6
4	112	13	12
6	680	20	18
8	6384	39	25
10	47127	93	39
12	364404	227	73
14	3473941	539	113
16	–	1301	211
18	–	3056	363
20	–	7276	676
22	–	18094	1219
24	–	39135	1641

Sacrificando Optimalidad Gradualmente

- A* con pesos (*weighted A**) es una buena opción cuando se está dispuesto a **sacrificar** optimalidad para obtener un mejor rendimiento.
- Consiste en usar A* con la siguiente función de evaluación

$$f(n) = g(n) + w \cdot h(n),$$

con $w \geq 1$.

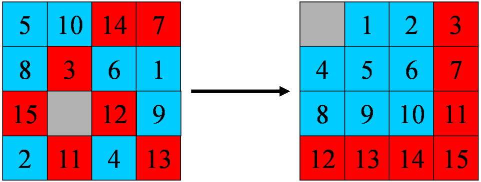
Teorema

Si h es admisible, weighted A* encuentra una solución cuyo costo es a lo más w veces el óptimo.

En la práctica encuentra soluciones mejores.

Pattern Databases

- Técnica para computar heurísticas admisibles a problemas difíciles.
- Consiste en construir una *abstracción* del problema de búsqueda.
- Se pre-computan soluciones óptimas para las abstracciones.



- En tiempo de búsqueda se usa el costo de esta solución como heurística.

Iterative Deepening A* – IDA*

- Algoritmo similar a A* pero mucho más eficiente en memoria
- Realiza una serie de búsquedas usando DFS.
- Se poda una rama cuando se excede un límite (threshold) de costo.
- El threshold inicial es el valor-h del nodo raíz.

Pseudo-code for IDA* (Edelkamp, 2011)

Procedure IDA*-Driver
Input: Implicit problem graph with start node s, weight function w, heuristic h, successor generation function *Expand*, and goal predicate *Goal*
Output: Path from s to $t \in T$, or $\emptyset$ if no such path exists

```
U' ← h(s)                                   ;; Initialize global threshold
bestPath ← ∅                                ;; Initialize solution path
while (bestPath = ∅ and U' ≠ ∞)             ;; Goal not found, unexplored nodes left
   U ← U'                                   ;; Reset global threshold
   U' ← ∞                                   ;; Initialize new global threshold
   bestPath ← IDA*(s, 0, U)                 ;; Invoke Alg. 5.8 at s
return bestPath                             ;; Terminate with solution path
```

Algorithm 5.7

Driver loop for IDA*.

Pseudo-code for IDA* (Edelkamp, 2011)

Procedure IDA*
Input: Node u, path length g, upper bound U
Output: Shortest path to a goal node $t \in T$, or $\emptyset$ if no such path exists
Side effects: Update of threshold U'

```
if (Goal(u)) return Path(u)                          ;; Terminate search
Succ(u) ← Expand(u)                                  ;; Generate successor set
for each v in Succ(u)                                ;; For all successors
  if (g + w(u,v) + h(v) > U)                         ;; Cost exceeds old bound
    if (g + w(u,v) + h(v) < U')                      ;; Cost smaller than new bound
      U' ← g + w(u,v) + h(v)                         ;; Update new bound
  else                                               ;; f-value below current threshold
    p ← IDA*(v, g + w(u,v), U)                       ;; Recursive call
    if (p ≠ ∅) return (u,p)                          ;; Solution found
return ∅                                             ;; No solution exists
```

Algorithm 5.8

The IDA* algorithm (no duplicate detection).