Razonamiento y Planificación Automática

César Augusto Guzmán Álvarez

Doctor en Inteligencia Artificial

Tema 6 : Búsqueda heurística

Sesión 1 / 2

unir LA UNIVERSIDAD EN INTERNET

Universidad Internacional de La Rioja

Resumen – Tema anterior

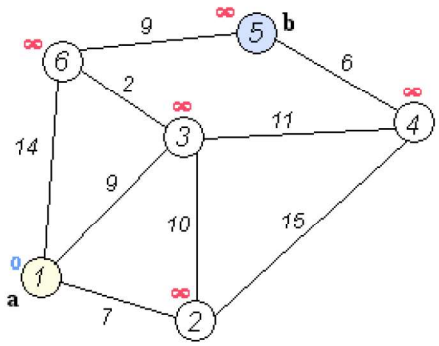
Tema 5 : Búsqueda offline

Sesión 1 :

- Agentes basados en búsqueda
- Búsqueda offline
- Búsqueda en amplitud

Sesión 2 :

- Búsqueda en profundidad
- Búsqueda de coste uniforme
- Practica – DFS and BFS



Índice

Sesión 1 :

- Que es una heurística ?
- Búsqueda A*

Sesión 2 :

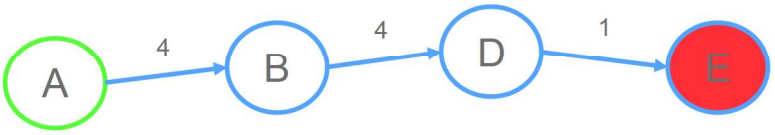
- Búsqueda por subobjetivos
- Búsqueda online

Que es heurística?

Definición de heurística: es la manera de alcanzar la solución de problemas a través de la **evaluación** de los progresos alcanzados durante la búsqueda del resultado objetivo.

h'(n) Coste **estimado** de alcanzar el objetivo

h(n) Coste **real** de alcanzar el objetivo



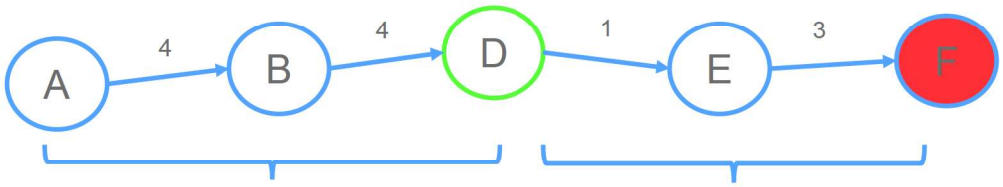
h'(n) = 3 h(n) = 9

$\forall n \in V$ Admisible : h'(n) <= h(n)

- Optimistas
- Solución óptima

No Admisible : h'(n) > h(n)

Búsqueda A*



g(n)
Coste real de llegar al nodo actual

h'(n)
Coste estimado de alcanzar el objetivo

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

→ Primero en profundidad
+
Primero en anchura

Dos estructuras:

- Abiertos – Cola de prioridades – ordenada por f(n)
- Cerrados – nodos ya visitados

Búsqueda A*

Propiedades:

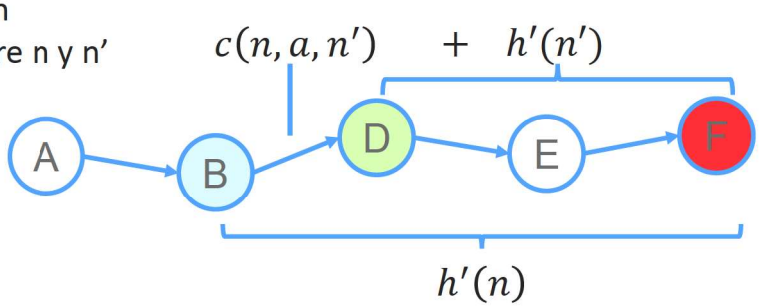
- Completo
- If $\forall\, n \in V,\ h'(n) = 0$: A* se comporta igual que la búsqueda de Coste Uniforme
- If $\forall\, n \in V,\ g(n) = 0$: A* se comporta igual que una búsqueda voraz (greedy)
- Para garantizar **admisibilidad** se debe cumplir que:

$$\forall\, n \in V, \qquad h'(n) \leq h(n)$$

- **Consistente** o monótona :

$$\forall\, n \in V\ ,\ h'(n) \leq c(n, a, n') + h'(n')$$

Donde:
n' = sucesores o hijos de n
a = acción o conector entre n y n'



Búsqueda A*

Algoritmo

// inicialmente todos los valores de g y f en los nodos son infinitos
Input: Grafo, Estado inicial S0, Estado final G

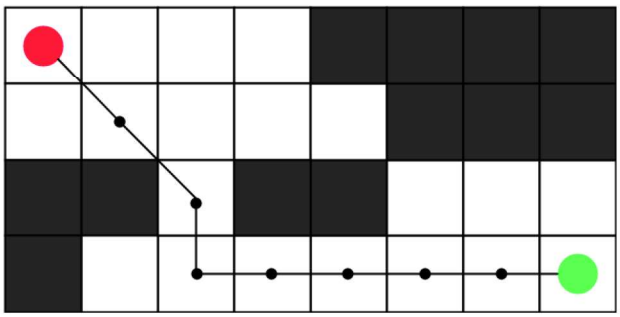
```
1:  Definir colaAbierta as colaPrioridad
2:
3:  S0.g ← 0
4:  S0.f ← h(Grafo, S0, G)
5:  colaAbierta ← {S0}
6:
7:  mientras colaAbierta ≠ ∅
8:      nodo ← extrae el de menor f de colaAbierta
9:      si G ⊆ nodo
10:         retornar camino a nodo
11:     fin si
12:     sucesores ← expandir(nodo)
13:     para cada sucessor ∈ sucesores hacer
14:         tentative_g ← nodo.g + c(nodo, sucessor)
15:         si tentative_g < sucessor.g
16:             sucessor.padre ← nodo
17:             sucessor.g ← tentative_g
18:             sucesor.f ← sucesor.g + h(Grafo, sucesor, G)
19:             si sucesor no en colaAbierta
20:                 colaAbierta ← colaAbierta ∪ sucesor
21:
22: retorna plan vacío o problema sin solución
```

nodo

- g = $infinito$
- f = $infinito$
- parent = null

A. Calcular el valor exacto (consume mucho tiempo)

1. Precalculando la distancia exacta entre cada nodo.
2. Si no hay obstaculos, podemos usar las fomulas de distancia (distancia euclidea).



Búsqueda A*

Algoritmo

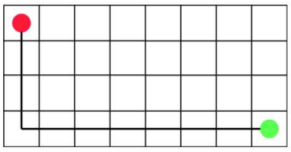
```
// initially all the values of g and f in the nodes are infinite
Input: (Graph, start, goal)

1:  Define openSet as priorityQueue
2:
3:  start.g = 0
4:  start.f = h(Graph, start, goal)
5:  openSet = {start}
6:
7:  while openSet is not empty
8:     current = openSet.Remove() // lowest f score value
9:     if current = goal
10:       return reconstruct_path(current)
11:
12:    for each neighbor of current
13:       tentative_g = current.g + c(current, neighbor)
14:       if tentative_g < neighbor.g
15:          neighbor.parent = current
16:          neighbor.g = tentative_g
17:          neighbor.f = neighbor.g + h(Graph, neighbor, goal)
18:          if neighbor not in openSet
19:             openSet.add(neighbor)
20:
21: return failure // the goal was never reached
```

nodo

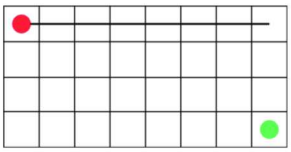
- g = $infinito$
- f = $infinito$
- parent = null

B. Aproximar el valor de h con alguna heuristica (consume menos tiempo).

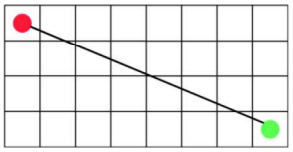
Manhattan Distance:
h = abs (current.x – goal.x) + abs (current.y – goal.y)



Diagonal Distance:
h = max { abs(current.x – goal.x), abs(current.y – goal.y) }



Euclidean Distance:
h = sqrt ((current.x – goal.x)2 + (current.y – goal.y)2)



Búsqueda A*

Algoritmo

```
// initially all the values of g and f in the nodes are infinite
Input: (Graph, start, goal)

1:   Define openSet as priorityQueue
2:
3:   start.g = 0
4:   start.f = h(Graph, start, goal)
5:   openSet = {start}
6:
7:   while openSet is not empty
8:      current = openSet.Remove() // lowest f score value
9:      if current = goal
10:        return reconstruct_path(current)
11:
12:     for each neighbor of current
13:        tentative_g = current.g + c(current, neighbor)
14:        if tentative_g < neighbor.g
15:           neighbor.parent = current
16:           neighbor.g = tentative_g
17:           neighbor.f = neighbor.g + h(Graph, neighbor, goal)
18:           if neighbor not in openSet
19:              openSet.add(neighbor)
20:
21:  return failure // the goal was never reached
```

nodo

- g = $infinito$
- f = $infinito$
- parent = null

Complejidad Computacional

O(E), donde E es el número de conexiones en el grafo.

Complejidad espacial

O(V), donde V es el número total de nodos.

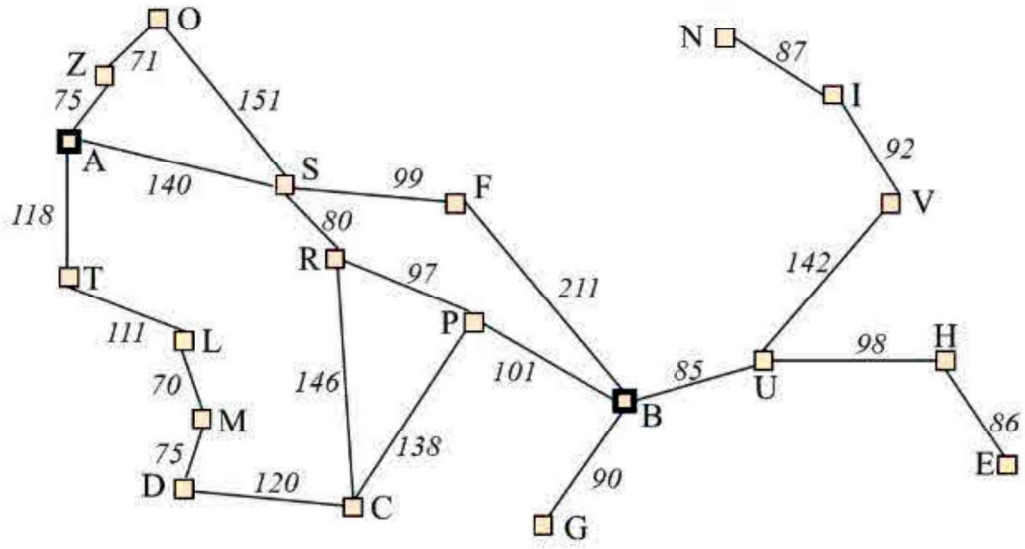
Algoritmo A*

// inicialmente todos los valores de g y f en los nodos son infinitos
Input: Grafo, Estado inicial S0, Estado final G

```
1:  Definir colaAbierta como colaPrioridad
2:  Definir listaCerrada como lista
3:  S0.g ← 0
4:  S0.f ← h(Grafo, S0, G)
5:  colaAbierta ← {S0}
6:  listaCerrada ← { }
7:  mientras colaAbierta ≠ ∅
8:      nodo ← extrae el de menor f de colaAbierta
9:      si G ⊆ nodo
10:         retornar camino a nodo
11:     fin si
12:     sucesores ← expandir(nodo)
13:     para cada sucesor ∈ sucesores hacer
14:         tentativa_g ← nodo.g + c(nodo, sucesor)
15:         si tentativa_g < sucessor.g
16:             sucesor.padre ← nodo
17:             sucesor.g ← tentativa_g
18:             sucesor.f ← sucesor.g + h(Grafo, sucesor, G)
19:             si sucesor en listaCerrada con mejor valor de f
20:                 continuar
21:             fin si
22:             colaAbierta ← colaAbierta ∪ sucesor
23:     fin para
24:     listaCerrada ← listaCerrada ∪ nodo
25: fin mientras
26: retorna plan vacío o problema sin solución
```

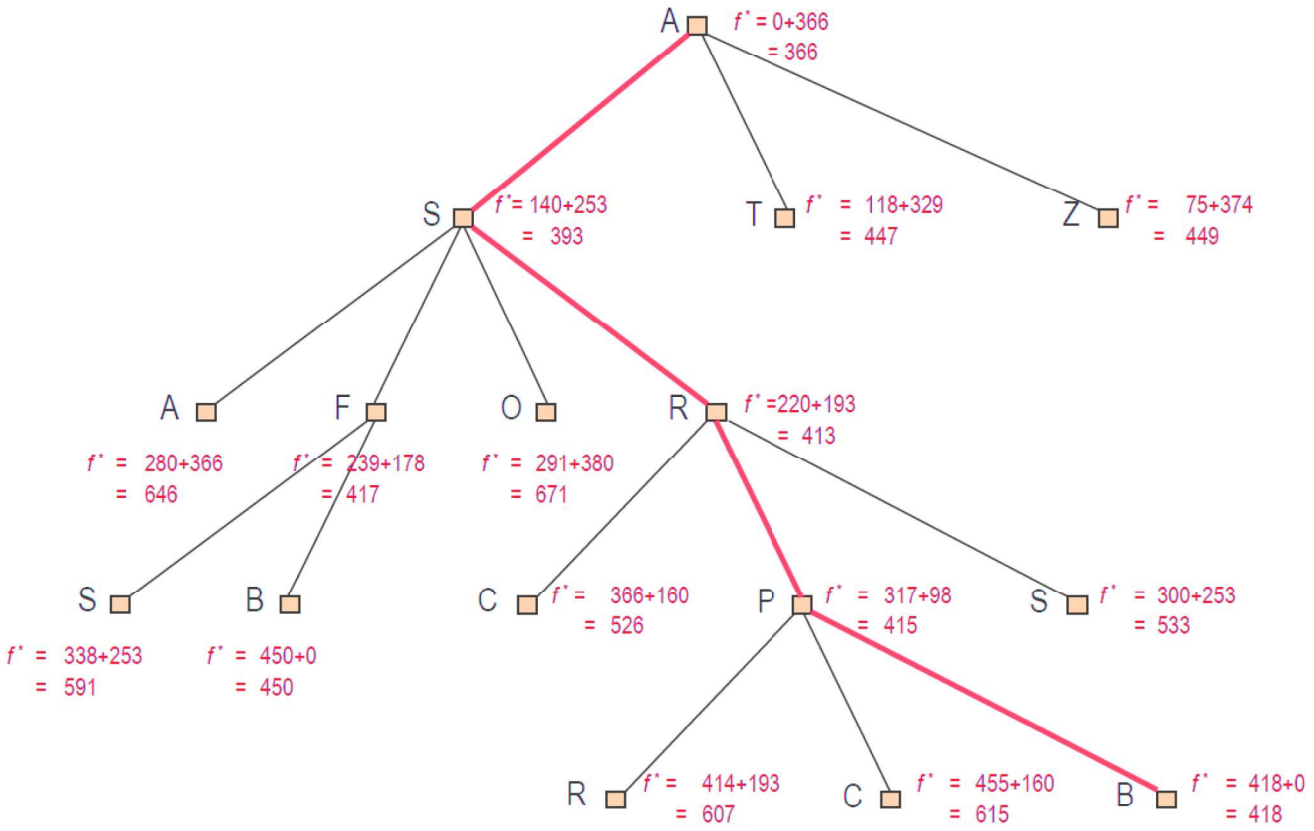
Búsqueda A*

Ejemplo



	h'
A	366
B	0
C	160
D	242
E	161
F	178
G	77
H	151
I	226
L	244
M	241
N	234
O	380
P	98
R	193
S	253

Búsqueda A*



	h'
A	366
B	0
C	160
D	242
E	161
F	178
G	77
H	151
I	226
L	244
M	241
N	234
O	380
P	98
R	193
S	253

Practica - Búsqueda A* :11-Puzzle.

Fuente: https://www.codingame.com/ide/puzzle/11-puzzle

Practica - Búsqueda A* :11-Puzzle.

Es un 3x4 puzzle con un cuadro vacío.
El objetivo es colocar los cuadros en sus posiciones correctas.
Para esto se deben ir moviendo los cuadros.
Solo se pueden mover los cuadros que están al lado del vacío.

índice fila y columna	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11

Estado final

Practica - Búsqueda A* :11-Puzzle.

Es un 3x4 puzzle con un cuadro vacío. El objetivo es colocar los cuadros en sus posiciones correctas. Para esto se deben ir moviendo los cuadros. Solo se pueden mover los cuadros que están al lado del vacío.

índice fila y columna	0	1	2	3
0	1	5	2	3
1	4	6	10	7
2	8	0	9	11

Acción: 2 0

índice fila y columna	0	1	2	3
0	1	5	2	3
1	4	6	10	7
2	0	8	9	11

Gracias!