Introducción a la Inteligencia Artificial
Trabajo Práctico 1: Representación y Búsqueda en Espacio de Estados

Agustín Fernández Bergé y Ramiro Gatto

14/04/2025

1. Ejercicio 1

Para implementar el algoritmo de DFS (Depth-First Search) utilizamos el algoritmo de búsqueda general, el cual es el siguiente:

Búsqueda General
    responde con **Solución** o **Falla**
    Lista-Nodo ← Estado Inicial

bucle hacer
    si Lista-Nodo está vacía contestar **Falla**
    tomo NODO de Lista-Nodo
    si NODO es **meta** contestar con NODO
    Lista-Nodo ← expansión NODO
FIN

En este caso, se colocan los NODOS generados al expandir, al comienzo de Lista-Nodos.

Para lograr el comportamiento de "agregar NODOS al comienzo de Lista-Nodo" lo que decidimos utilizar fue una pila, la cual captura esa idea de agregar al inicio y sacar del inicio. Es decir, Lista-Nodos es una pila.

De esta forma, combinado el pseudo código y la pila es como implementamos el algoritmo de DFS. Luego al usar esta solución con el *bigMaze* y con *PositionSearchProblem* se obtiene:

```
[SearchAgent] using function dfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.02 seconds
Search nodes expanded: 390
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```

2. Ejercicio 2

Para la implementación de BFS (Breadth-First Search) también utilizamos el algoritmo de búsqueda general, solo que en este se colocan los NODOS generados al expandir al final de Lista-Nodos.

Para este comportamiento se utilizo una cola (normal), la cual permite agregar al final y sacar del inicio. Es decir, Lista-Nodos es una pila. En este caso, nuevamente lo probamos con el *bigMaze* y con *PositionSearchProblem* obteniéndose:

```
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.02 seconds
Search nodes expanded: 620
Pacman emerges victorious! Score: 300
```

```
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```

3. Ejercicio 3

Para implementar UCS (Uniform Cost Search) nuevamente se utilizo el algoritmo de búsqueda general, solo que ahora al colocar los NODOS generados al expandir lo hacemos en base al costo del mismo. Agregando al inicio los de menor costo.

Para lograr esto se utilizan las colas de prioridad, siendo la prioridad el costo del nodo. Nuevamente probamos con el *bigMaze* y con *PositionSearchProblem*, teniéndose:

```
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.02 seconds
Search nodes expanded: 620
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```

4. Ejercicio 4

4.1. Búsqueda A*

Recordemos que el A* utiliza una función $f(Nodo)$ de evaluación de la forma: $f(n) = g(n) + h(n)$ donde,

- g(n) = costo hasta llegar a n
- h(n) = costo estimado hasta la meta desde n
- f(n) = costo total de ruta pasando por n hasta la meta

En este nuevamente utilizamos el algoritmo de búsqueda general en donde Lista-Nodos se ordena de acuerdo al valor de $f(Nodo)$.

Para este caso utilizamos nuevamente las colas de prioridad, y ademas también nos fijamos si la heurística que se utiliza es consistente, antes de hacer la búsqueda.

Ahora, probamos con el *bigMaze*, con *PositionSearchProblem* y utilizando como heurística la "distancia de manhattan". teniéndose:

```
[SearchAgent] using function astar and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.02 seconds
Search nodes expanded: 549
```

```
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```

En donde se puede observar que se corrobora la dicho en el enunciado, esto es que hay una diferencia de uno 100 nodos expandidos con respecto a UCS.

4.2. Laberinto openMaze

En esta sección vamos a analizar las 4 forma de búsqueda que hicimos con el laberinto **openMaze**. El recorrido que hacen es el siguiente:

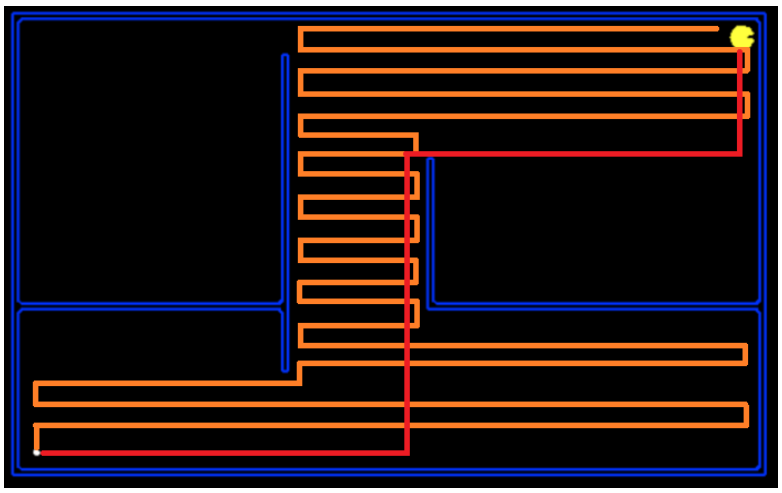


Figura 1: En naranja el camino del DFS, En rojo el camino del resto

Donde se tiene que todos menos el DFS siguen la misma ruta, pero exploran una cantidad de nodos distinto (535 en A*, 682 en UCS y en BFS, 576 en DFS).

A pesar de que A*, UCS y BFS expanden una cantidad distinta de nodos todos tienen el mismo puntaje (tardan lo mismo). Mas aun, a pesar de que DFS es uno de los que menos nodos expande este es el de peor puntaje (mas lento).

5. Ejercicio 5

5.1. cornersProblems

En este ejercicio tenemos que elegir una forma de representar el estado de tal forma que detecte si las 4 esquinas del laberinto fueron alanzadas. La forma que elegimos fue tomar al estado como: **(Posición actual, (Bool, Bool, Bool, Bool))**.

Es decir, el estado es una tupla cuyas componentes son:

- La posición actual.
- Una cuádrupla de Bool, cada uno representa una esquina del tablero en este orden: (1, 1), (1, top), (right, 1), (right, top). Al principio estan todas en False.

De esta forma cuando se llega a una esquina se cambia el False correspondiente por un True. Para ver si se alcanzaron todas las esquinas se hace un And generalizado entre todas las componentes de la cuádrupla.

5.2. Posibles heurísticas

Para el tema de las heurística se pensaron unas cuantas, y para ver si es admisible hay que tener en cuenta que la heurística no debe sobrestimar. La primera que se pensó fue una estilo greedy.

Punto-actual ← Posicion-inicial
Distancia ← 0

bucle hacer
    si Lista-Esquinas está vacía contestar **Distancia**
    tomo ESQUINA de Lista-Esquinas-No-Visitadas
    Distancia ← Distancia + distanciaManhattan(Punto-actual, ESQUINA)
FIN

Es decir, se calcula la distancia de manhattan entre el punto actual y la esquina mas cercana, y se suma a la distancia total. Luego se repite el proceso hasta que no haya mas esquinas. Esta heurística es valida pero puede elegir rutas no optimas en ciertas situaciones(por ejemplo, en un escenario donde no hay paredes), por lo que no es admisible.

La siguiente forma fue considerar la cantidad de esquinas no visitadas. Esta heurística es consistente, ya que la cantidad de esquinas no visitadas en cada paso disminuye en como mucho una unidad, por lo tanto es admisible. Sin embargo la estimación del costo a la solución es demasiado optimista, por lo que la cantidad de nodos expandidos disminuye muy poco.

También se intento repetir la primera heurística pero usando la distancia euclidea y la distancia con la norma infinito. En algunos laberintos no se cumplió la propiedad de consistencia usando la distancia euclidea pero si se mantuvo usando la distancia con norma infinito. Sin embargo es posible que haya algunos layouts donde existan un par de estados donde la propiedad de consistencia se rompa, por lo que descartamos esta heurística.

La ultima heurística que pensamos fue, a partir de la posición actual, revisar todas las formas posibles de visitar las esquinas, calcular la distancia recorrida total de cada recorrido (usando distancia de manhattan) y devolver la menor de todas ellas.

Para una posición inicial S solamente hay que visitar 4 esquinas, por lo que solo hay que revisar $4! = 24$ posibles ordenes. Además, se pueden memorizar algunos resultados para mejorar la velocidad de computo de la heurística (aunque esto también gasta más memoria). Esta heurística es la resolución de una restricción del problema a cuando no hay paredes en el laberinto, asi que es esperable que sea consistente y de hecho lo es. La heurística es consistente ya que en cada paso la distancia a la solución varia en a lo sumo 1 paso, dado que el costo de ir de un nodo a otro es de 1, la heurística cumple la propiedad de consistencia. Además disminuye mucho la cantidad de nodos explorados (741 nodos para la solución optima vs 1966 nodos usando búsqueda de costo uniforme).

6. Ejercicio 6

La heurística no trivial consistente elegida para resolver el problema de comer todas las pastillas en las esquinas es equivalente a aquella que revisa todos los ordenes para visitar las esquinas y elige el menor, solo que es mas eficiente ya que evita revisar algunos recorridos no óptimos.

Resulta que la primera heurística que siempre intenta visitar la esquina mas cercana obtiene la respuesta optima a visitar las esquinas de un tablero sin paredes cuando la casilla de partida es una de las esquinas. Vimos que esto es asi analizando el escenario de un laberinto sin paredes donde hay que visitar 3, 2 o 1 esquina.

Si S es la posición inicial y A, B, C y D son las esquinas entonces los recorridos posibles entran en alguno de estos 4 esquemas:

- $S \to A \to \ldots$
- $S \to B \to \ldots$
- $S \to C \to \ldots$
- $S \to D \to \ldots$

Los subrecorridos que siguen desde las esquinas A, B, C y D pueden resolverse optimamente usando la primera heurística, por lo que solo hay que revisar cuatro recorridos posibles.

Asi que la heurística final consiste en el siguiente algoritmo:

Distancia $\leftarrow \infty$

PARA Esquina EN Lista-Esquinas-No-Visitadas
    Lista' $\leftarrow$ Lista-Esquinas
    ELIMINAR Esquina de Lista'
    Distancia $\leftarrow$ MINIMO(Distancia, distanciaManhattan(Posicion-Actual, Esquina) + busquedaGreedy(Lista'))
FIN

Donde busquedaGreedy es la heurística que siempre busca visitar la esquina mas cercana, calculando distancias usando la distancia de manhattan.

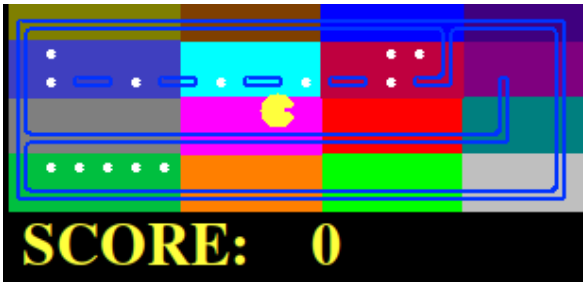
7. Ejercicio 7

La heurística elegida para ejercicio 7 consiste en dividir el tablero en cuadriculas no superponibles (en el código son 16 cuadriculas, pero puede ser cualquier potencia de 2). En esta division posiblemente halla cuadriculas que no tengan pastillas ya otras que si. A la hora de calcular la heurística, se calcula el largo del mínimo camino que visita todas las cuadriculas que tienen pastillas (ignorando las paredes del laberinto).

Esta heurística surgió de la idea de generalizar el problema de visitar las 4 esquinas del tablero al problema de comer todas las pastillas. En este caso cada cuadricula representa una

"esquina" a visitar, solo que las cuadriculas ocupan mas espacio.

Para mejorar la heurística, se restringe el tamaño de las cuadriculas para que haya al menos dos esquinas opuestas que tengan pastillas. Además algunos resultados pueden calcularse más de una vez por lo que se memorizan los resultados y calculados para mejorar la velocidad de computo(a costa de usar mas memoria). La heurística se acerca mas a la solución real a medida que se aumentan las cuadriculas, pero también aumenta el tiempo de computo y la memoria utilizada.



(a) Por ejemplo, para el layout trickySearch, el laberinto se divide en 16 cuadriculas.

(b) Si se eliminan las cuadriculas sin pastillas y se restringe el tamaño de las cuadriculas restantes, entonces solo quedan 4 cuadriculas.

El mínimo camino que visita todas las cuadriculas consta de 17 pasos, mientras que el costo real de comer todas las pastillas es de 60 pasos. Usando esta heurística el agente encuentra la solución en aproximadamente 0.8 segundos explorando 9086 nodos.

La heurística es admisible porque un recorrido que tiene que comer todas las pastillas como mínimo tiene que visitar todas las cuadriculas (y siempre donde hay una cuadricula hay por lo menos 1 pastilla). Además es consistente porque en cada paso la distancia a la solución varia en solo 1 paso.