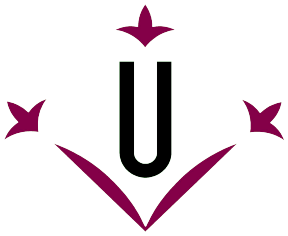

Universitat de Lleida
Escola Politècnica Superior

Inteligencia Artificial

Práctica 1 (reentrega): *Pacman*

Pere Rollón Baiges
Jordi Rafael Lazo Florensa

16 de febrero de 2021

1 Algoritmo A* (búsqueda en grafo)

Tal y como se puede observar en la Figura 1 para al implementación de este algoritmo nos hemos basado en el algoritmo UCS que implementamos en clase de Inteligencia Artificial.

En el A* la prioridad de cada nodo se calcula mediante la fórmula $f(n) = g(n) + h(n)$ dónde $f(n)$ es la función de evaluación o la prioridad con que se introduce el nodo a la cola con prioridad.

La modificación que se ha hecho frente el UCS consiste en sumar la heurística $h(n)$, que es el coste estimado desde el nodo actual hasta el objetivo, la cual esta compuesta por $n.state$ que es el estado del nodo actual y *problem* que es el problema que se esta tratando, al coste del camino desde el inicio hasta el nodo $g(n)$ que es $n.cost$.

Para que A* retornase una solución con coste óptimo las heurísticas deben ser consistentes.

```python
def uniformCostSearch(problem):
    """Search the node of least total cost first."""
    "*** YOUR CODE HERE ***"
    generated = {}
    fringe = util.PriorityQueue()
    n = Node(problem.getStartState())
    fringe.push(n,n.cost)
    generated[n.state] = [n,'F']

    while True:
        if fringe.isEmpty():
            print("No solution")
            sys.exit(-1)
        n = fringe.pop()
        if problem.isGoalState(n.state):
            return n.total_path()
            if generated[ns.state][1] == 'E':
                continue

        generated[n.state] = [n,'E']
        for state, action, cost in problem.getSuccessors(n.state
            ns = Node(state,n,action,n.cost + cost)
            if not ns.state in generated:
                fringe.push(ns, ns.cost)
                generated[ns.state] = [ns,'F']
            elif ns.cost < generated[ns.state][0].cost:
                fringe.push(ns, ns.cost)
                generated[ns.state] = [ns,'F']
```

```python
def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic
    "*** YOUR CODE HERE ***"
    generated = {}
    fringe = util.PriorityQueue()
    n = Node(problem.getStartState())
    fringe.push(n, n.cost + heuristic(n.state,problem))
    generated[n.state] = [n, 'F']
    while True:
        if fringe.isEmpty():
            print "NO SOLUTION"
            sys.exit(-1)
        n = fringe.pop()
        if problem.isGoalState(n.state):
            return n.total_path()
        if generated[n.state][1] == 'E': continue
        generated[n.state] = [n, 'E']
        for state, action, cost in problem.getSuccessors(n.state):
            ns = Node(state, n, action, n.cost + cost)
            if ns.state not in generated:
                fringe.push(ns,  ns.cost + heuristic(ns.state,problem))
                generated[ns.state] = [ns, 'F']
            elif ns.cost < generated[ns.state][0].cost:
                fringe.push(ns, ns.cost + heuristic(ns.state,problem))
                generated[ns.state] = [ns, 'F']
```

Figura 1: En la izquierda la implementación del algoritmo UCS y en la derecha la implementación del A*.

La distancia euclidiana es un número positivo que indica la separación que tienen dos puntos en un espacio donde se cumplen los axiomas y teoremas de la geometría de Euclides.

Se define como:

$$d_E(P_1, P_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

```
[SearchAgent] using function astar and heuristic euclideanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 557
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Figura 2: Resultado obtenido del A* con la heurística distancia Euclídea.

Por lo que concierne a la distancia Manhattan, esta nos da la distancia mínima entre dos coordenadas de una forma parecida a la utilizada en la distancia euclidiana. Esta distancia entre dos puntos se calcula como la longitud de cualquier camino que los una mediante segmentos verticales y horizontales; todos miden lo mismo.

Se define como:

$$d_1(p, q) = \|p - q\|_1 = \sum_{i=1}^{n} |p_i - q_i|,$$

```
[SearchAgent] using function astar and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 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
```

Figura 3: Resultado obtenido del A* con la heurística distancia de Manhattan.

Ya que los dos algoritmo han sido ejecutados en el mapa *bigCorners* la disposición de este mapa y los muros ha facilitado la búsqueda con la heurística distancia Manhattan en comparación con la distancia Euclídea. Este es el motivo principal por el cual la los nodos expandidos con la $h(n)$ Manhattan son de 549 mientras que con la $h(n)$ Euclídea son de 557.

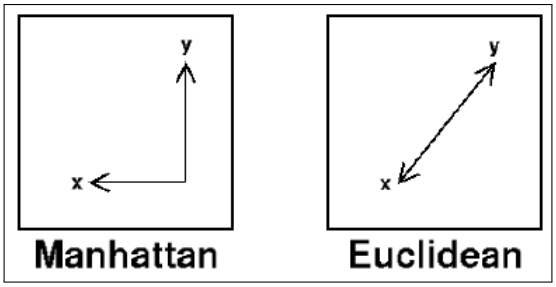

Figura 4: Diferencia entre la distancia Manhattan y Euclíidea.

2 Implementación Heurística para *Corners Problem*

Para la implementación del método *cornersHeuristic* se ha analizado como eran los estados de la clase *cornersProblem*.

Como se puede observar en la Figura 5, cada estado está formado por la posición del pacman y las cuatro posiciones de la comida (una por cada esquina).

Para calcular la heurística se ha tomado en cuenta la comida en cada esquina. En el caso que no haya comida o se haya terminado porque el pacman se la ha comido, la heurística tendrá el valor de 0.

En el caso que existiera comida, se obtiene la posición actual del pacman y la distancia Manhattan correspondiente para cada esquina y a medida que se calculan las distancias se almacena la distancia mayor de acuerdo a un criterio.

Este criterio compara la distancia mayor guardada y la nueva obtenida, en el caso que la nueva distancia fuese mayor esta será guardada sustituyendo a la anterior.

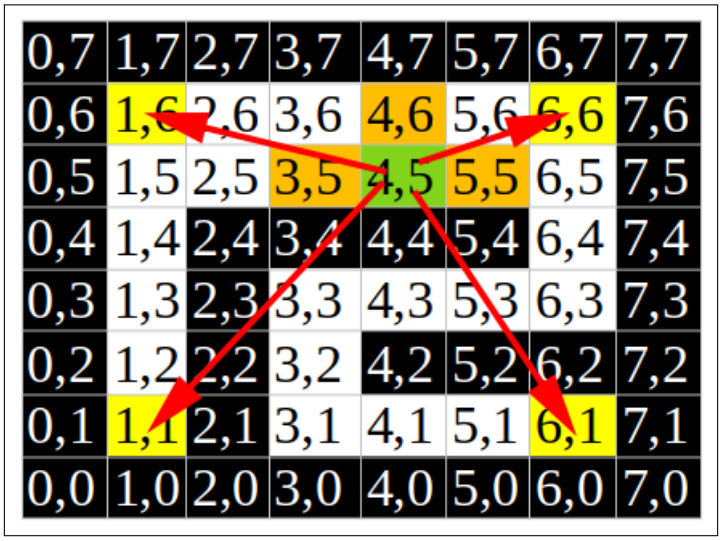

Figura 5: Representación gráfica del mapa *tinyCorners*.

3 Implementación Heurística para *Food Search Problem*

En la primera entrega de esta práctica, en este apartado se optó por una estrategia similar en el problema *Corners Problem*, puesto que *Food Search Problem* es una generalización de dicho problema. La diferencia residia en que en este método la distancia se obtenia mediante la función *mazeDistance* que calculaba la distancia entre dos puntos a traves de los costes de paso utilitzando el algoritmo BFS. Finalmente, la lista de posiciones de comida dejaban de ser esquinas para ser cualquier punto del mapa.

Esta primera solución funcionaba correctamente pero el timepo de ejecución era superior al esperado para una heuristica.

Por eso en esta segunda implementación se ha optado por utilizar el diccionario propio del juego llamado *problem.heuristicInfo*, para el calculo de distancias se ha utilizando la misma estratégia pero antes de calcular la distancia entre el pacman y la comida se ha consultado si esa heuristica ya se había calculado.