Universidad Autónoma de Madrid
Escuela Politéctica Superior

Inteligencia Artificial

Práctica 1: Búsqueda

Pareja 4 (2301): Leandro García y Fabián Gutiérrez

8 de marzo de 2021

Sección 1

1. Como se discutió en teoría, los algoritmos de búsqueda estudiados son idénticos salvo la estructura de datos que almacena los nodos abiertos, es decir, salvo el orden en el que se expanden los nodos generados. Por tanto, la idea inicial fue crear una función genérica que recibiera en sus argumentos la estructura de datos de la lista de nodos abiertos. De este modo, la búsqueda primero en profundidad con eliminación de estados repetidos se reduce a llamar a esta función genérica indicando que la estructura a utilizar es una pila.

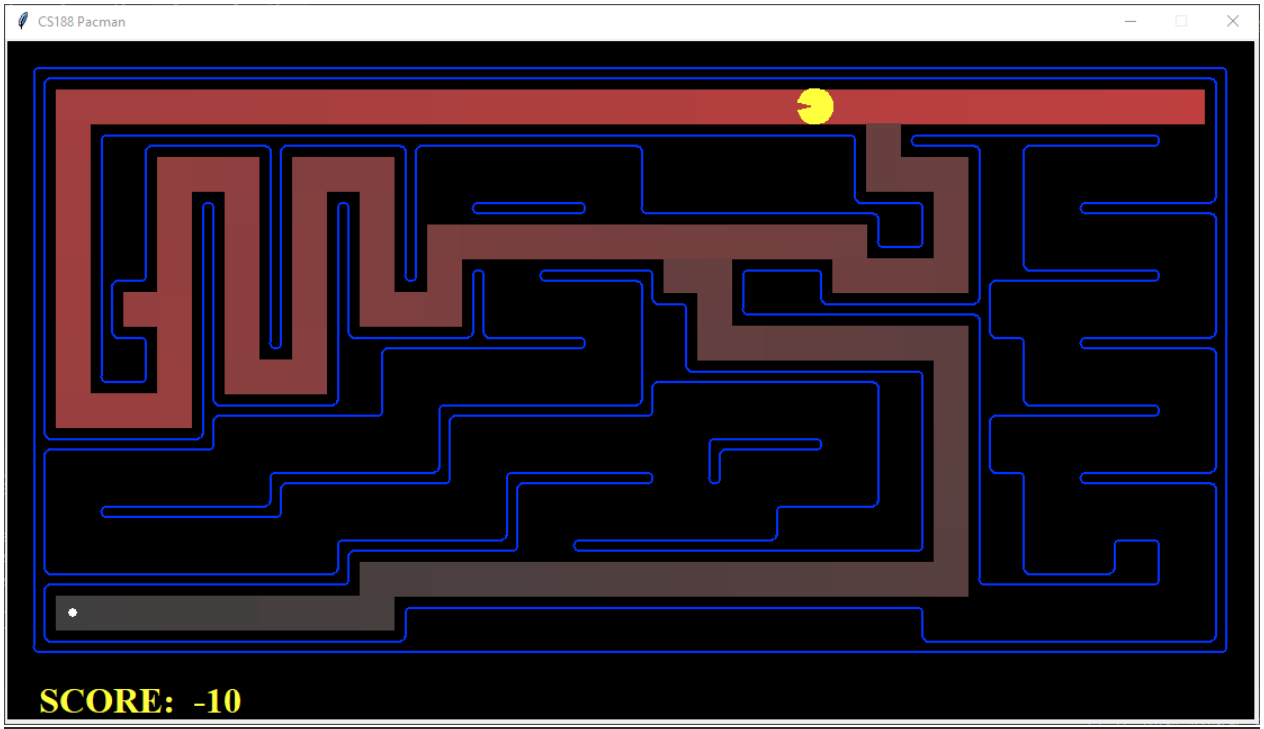
2. Para esta primera versión de `generalSearch()`, las funciones del framework utilizadas son los métodos de la clase `SearchProblem`: `getStartState()` para obtener el nodo de partida, `getSuccessors()` para generar los sucesores del nodo en expansión e `isGoalState()` para comprobar si la búsqueda ha finalizado con éxito. Además, `listType` debe ser una de las estructuras de datos definidas en `util.py`, ya que todas implementan la interfaz requerida en `generalSearch` (i.e. los métodos `pop()` y `push()`) y además abarcan las diferentes estructuras propias de cada algoritmo de búsqueda a implementar. Para `depthFirstSearch()`, el tipo es `util.Stack`.

3. Además de los argumentos comentados anteriormente, se añade la bandera `enableRepetition` para poder implementar búsquedas sin eliminación de estados repetidos, aunque en esta práctica nunca sea el caso.

```python
def depthFirstSearch(problem):
    return generalSearch(problem, util.Stack)


def generalSearch(problem, listType, enableRepetition=False):
    abiertos = listType()
    cerrados = dict()
    ret = []

    s = problem.getStartState()
    if problem.isGoalState(s):
        return ret

    cerrados[s] = None

    for t in problem.getSuccessors(s):
        abiertos.push((t, s))

    while not abiertos.isEmpty():
        s = abiertos.pop()
        if s[0][0] not in cerrados.keys() or enableRepetition:
            cerrados[s[0][0]] = (s[0][1], s[1])
            if problem.isGoalState(s[0][0]):
                break
            for t in problem.getSuccessors(s[0][0]):
                abiertos.push((t, s[0][0]))

    s = s[0][0]
    m = cerrados[s]
    while m is not None:
```

```
            ret.append(m[0])
            s = m[1]
            m = cerrados[s]

        ret.reverse()
        return ret
```

4. A continuación se muestran los resultados y nodos expandidos de la búsqueda en profundidad para `mediumMaze`:




```
C:\Users\leand\Documents\GIT\IA_2021\P1>python pacman.py -l mediumMaze -p SearchAgent
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 130 in 0.0 seconds
Search nodes expanded: 146
Pacman emerges victorious! Score: 380
Average Score: 380.0
Scores:        380.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Figura 1: Resultados de la búsqueda en profundidad

5. La búsqueda primero en profundidad, en general, no es óptima. Por ejemplo, para la configuración `mediumMaze` sí alcanza una solución de coste 130 expandiendo 146 nodos, pero, como se verá más adelante, esta no es óptima.

6. **¿El orden de exploración es el que esperabais? ¿Pacman realmente va a todas las casillas exploradas en su camino hacia la meta?**

   El orden de exploración de los nodos es el esperado para el algoritmo implementado. Además de priorizar la profundidad antes que la anchura a la hora de elegir un nodo para expandir, el algoritmo toma el último de los nodos visitados en caso de disponer de múltiples opciones. Esto último se debe a la implementación mediante una estructura de pila, con una política LIFO (*Last In, First Out*).

   Una estrategia alternativa hubiese sido la utilización de un algoritmo recursivo, lo que implicaría que los nodos de igual profundidad serían tomados en orden de generación, al contrario que en el método escogido.

7. **¿Es esta una solución de menor coste? Si no es así, pensad qué está haciendo mal la búsqueda en profundidad.**

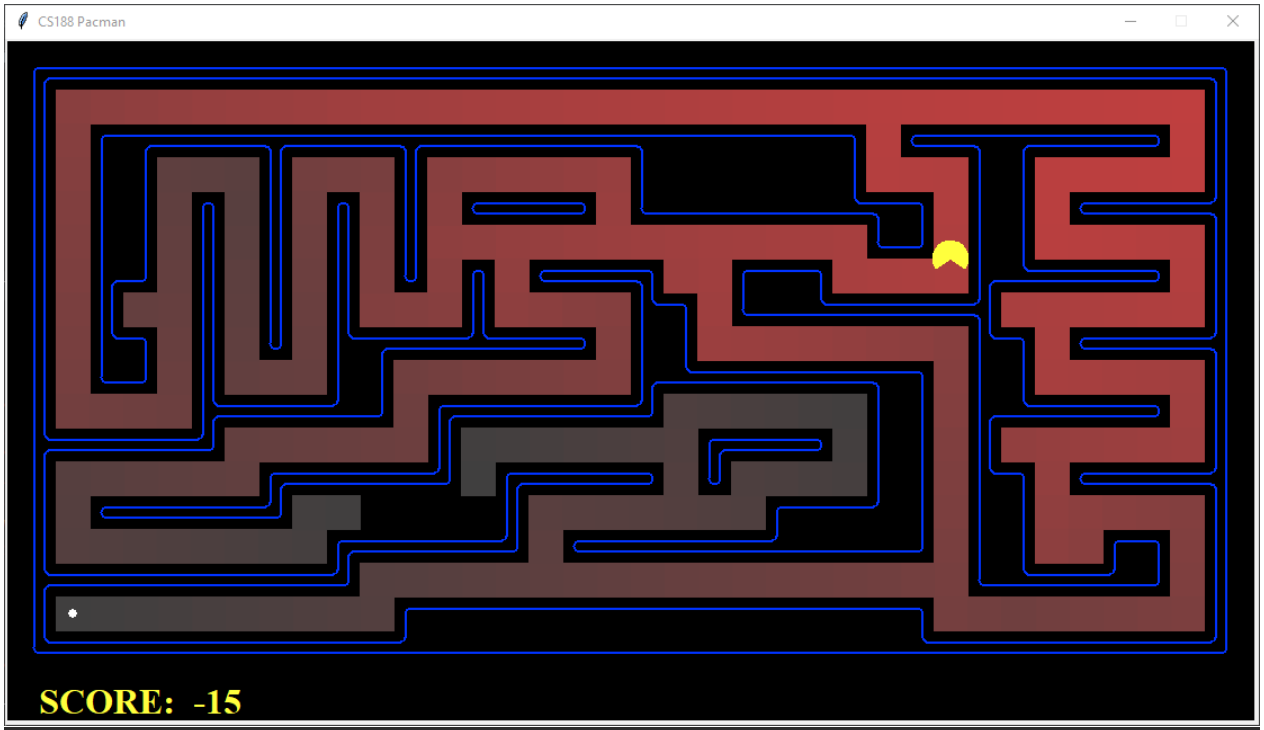
   La búsqueda en profundidad junto con la eliminación de estados repetidos, es un algoritmo completo (siempre haya solución). Sin embargo, este toma la primera solución que se encuentre, y al priorizarse la profundidad puede estar descartando otra con un coste menor. De hecho, como ya se ha mencionado anteriormente, la solución puede variar en función de la implementación del propio algoritmo.

Sección 2

1. La anterior implementación de `generalSearch()` hace trivial la implementación del algoritmo de búsqueda en anchura: basta con indicar en los argumentos que la estrucutura a utilizar es una cola FIFO (*First In, First Out*).
2. Para este algoritmo, el argumento `listType` de `generalSearch()` es `util.Queue`, que como se discutió anteriormente implementa la interfaz necesaria.
3. La implementación de `generalSearch()` exhibida antes se mantiene intacta.

```
def breadthFirstSearch(problem):
    return generalSearch(problem, util.Queue)
```

4. En este apartado se muestran los resultados en `mediumMaze` de la búsqueda en anchura y la imagen que muestra la distribución de los nodos expandidos.




```
C:\Users\leand\Documents\GIT\IA_2021\P1>python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Figura 2: Resultados de la búsqueda en anchura

5. Como el coste es proporcional a la profundidad, la búsqueda primero en anchura es óptima y, en consecuencia, encuentra la mejor solución en todos los casos de prueba. Además, es completa, por lo que está garantizado (y de hecho se cumple) que encuentre siempre la solución. Por ejemplo, para la configuración `mediumMaze` llega a la solución óptima, de coste 68, expandiendo 68 nodos, lo que supone dos mejoras respecto a la búsqueda primero en profundidad, aunque la complejidad espacial es bastante peor a medida que aumenta el tamaño del problema.

6. **¿Encuentra la búsqueda en anchura una solución de menor coste?**

   Dado que la búsqueda se basa en dar prioridad a nodos de menor profundidad, el algoritmo asegura encotrar la solución menos profunda. La optimalidad del resultado se dará entonces cuando los costes sean no decrecientes con la profundidad.

   En el caso particular del problema de búsqueda de una posición del laberinto (`Position-`

`SearchProblem`) para laberintos sin fantasmas, el coste de todas las acciones es 1, por lo que el camino hallado es, en efecto, óptimo.

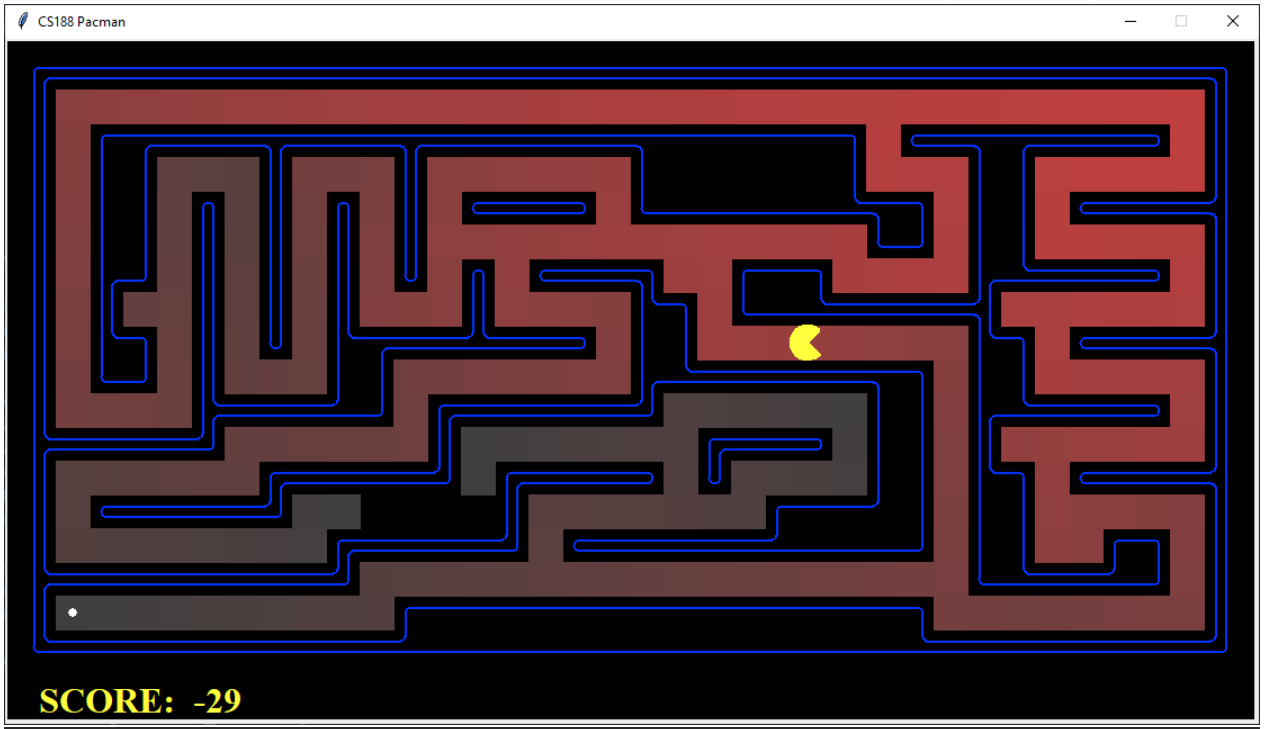
Sección 3

1. Para implementar la búsqueda de coste uniforme hubo que añadir a los estados de búsqueda de `generalSearch()` su coste acumulado, esto es, la suma de los costes de las acciones que llevan del nodo inicial a ese nodo. El coste acumulado de un nuevo estado de búsqueda es el coste acumulado de su predecesor más el coste de la acción en cuestión, partiendo de que el coste del nodo inicial es cero (por tanto, no hace falta recorrer todo el camino inverso hasta la raíz para computar este coste).
2. Una vez más, `util.py` cuenta con una cola de prioridad, que es la estructura de datos que usa la búsqueda de coste uniforme. Específicamente, `listType` en este caso es `util.PriorityQueueWithFunction` para poder asignar que la prioridad viene dada por el coste acumulado.
3. Como las otras estructuras de datos utilizadas no requieren argumentos en sus constructores pero `util.PriorityQueueWithFunction` sí, se añadió la bandera `priorityQueue` a los argumentos de `generalSearch()` para diferenciar este caso. Obsérvese que el valor por defecto de esta bandera es `False`, con lo que las implementaciones de los algoritmos de búsqueda previos siguen funcionando.

```
def uniformCostSearch(problem):
    return generalSearch(problem, util.PriorityQueueWithFunction,
        priorityQueue=True)


def generalSearch(problem, listType, priorityQueue=False, enableRepetition=False):
    if priorityQueue:
        abiertos = listType(lambda node: node[2])
    else:
        abiertos = listType()

    cerrados = dict()
    ret = []

    s = problem.getStartState()
    if problem.isGoalState(s):
        return ret

    cerrados[s] = None

    for t in problem.getSuccessors(s):
        abiertos.push((t, s, t[2]))

    while not abiertos.isEmpty():
        s = abiertos.pop()
        if s[0][0] not in cerrados.keys() or enableRepetition:
            cerrados[s[0][0]] = (s[0][1], s[1])
            if problem.isGoalState(s[0][0]):
                break
            for t in problem.getSuccessors(s[0][0]):
```

```
                abiertos.push((t, s[0][0], t[2] + s[2]))

    s = s[0][0]
    m = cerrados[s]
    while m is not None:
        ret.append(m[0])
        s = m[1]
        m = cerrados[s]

    ret.reverse()
    return ret
```

4. Aquí vemos el producto de aplicar búsqueda de coste uniforme al laberinto `mediumMaze`.




```
C:\Users\leand\Documents\GIT\IA_2021\P1>python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Figura 3: Resultados de la búsqueda en de coste uniforme

5. Como el coste de las acciones es estrictamente mayor que cero, era de esperar que la búsqueda

de coste uniforme fuera óptima (como de hecho lo es en las pruebas). Además, el coste de cada acción es 1, por lo que no puede haber caminos infinitos de coste finito, con lo que la búsqueda de coste uniforme es completa (de hecho, siempre llega a la solución en las pruebas). Por ejemplo, en la configuración de `mediumMaze` alcanza la solución óptima de coste 68 expandiendo 269 nodos, al igual que la búsqueda primero en anchura.

Sección 4

1. Para implementar A*, el último de los algoritmos solicitados, solo hizo falta añadir el valor de la heurística al coste acumulado a la función de prioridad de la cola usada para la búsqueda de coste uniforme.
2. Las funciones y estructuras utilizadas por A* son idénticas a las de la búsqueda de coste uniforme, con la diferencia de que la función de prioridad para `util.PriorityQueueWithFunction` incluye el valor de la heurística en el nodo a evaluar.
3. Finalmente, esta es la versión de `generalSearch()` que implementa, de forma general, todos los algoritmos solicitados. Cabe destacar que el valor por defecto de `heuristic` es la heurística idénticamente nula, con lo que la función de prioridad de los nodos es la utilizada en la búsqueda de coste uniforme (así que la definición de `uniformCostSearch()` anterior sigue funcionando).

```
def aStarSearch(problem, heuristic=nullHeuristic):
    return generalSearch(problem, util.PriorityQueueWithFunction, True, heuristic)


def generalSearch(problem, listType, priorityQueue=False,
    heuristic=nullHeuristic, enableRepetition=False):
    if priorityQueue:
        abiertos = listType(lambda node: node[2] + heuristic(node[0][0], problem))
    else:
        abiertos = listType()

    cerrados = dict()
    ret = []

    s = problem.getStartState()
    if problem.isGoalState(s):
        return ret

    cerrados[s] = None

    for t in problem.getSuccessors(s):
        abiertos.push((t, s, t[2]))

    while not abiertos.isEmpty():
        s = abiertos.pop()
        if s[0][0] not in cerrados.keys() or enableRepetition:
            cerrados[s[0][0]] = (s[0][1], s[1])
            if problem.isGoalState(s[0][0]):
                break
            for t in problem.getSuccessors(s[0][0]):
                abiertos.push((t, s[0][0], t[2] + s[2]))
```

```
    s = s[0][0]
    m = cerrados[s]
    while m is not None:
        ret.append(m[0])
        s = m[1]
        m = cerrados[s]

    ret.reverse()
    return ret
```

4. A continuación se pueden observar la salida del programa y el laberinto (con los nodos expandidos) para la búsqueda A* en el laberinto mediano.

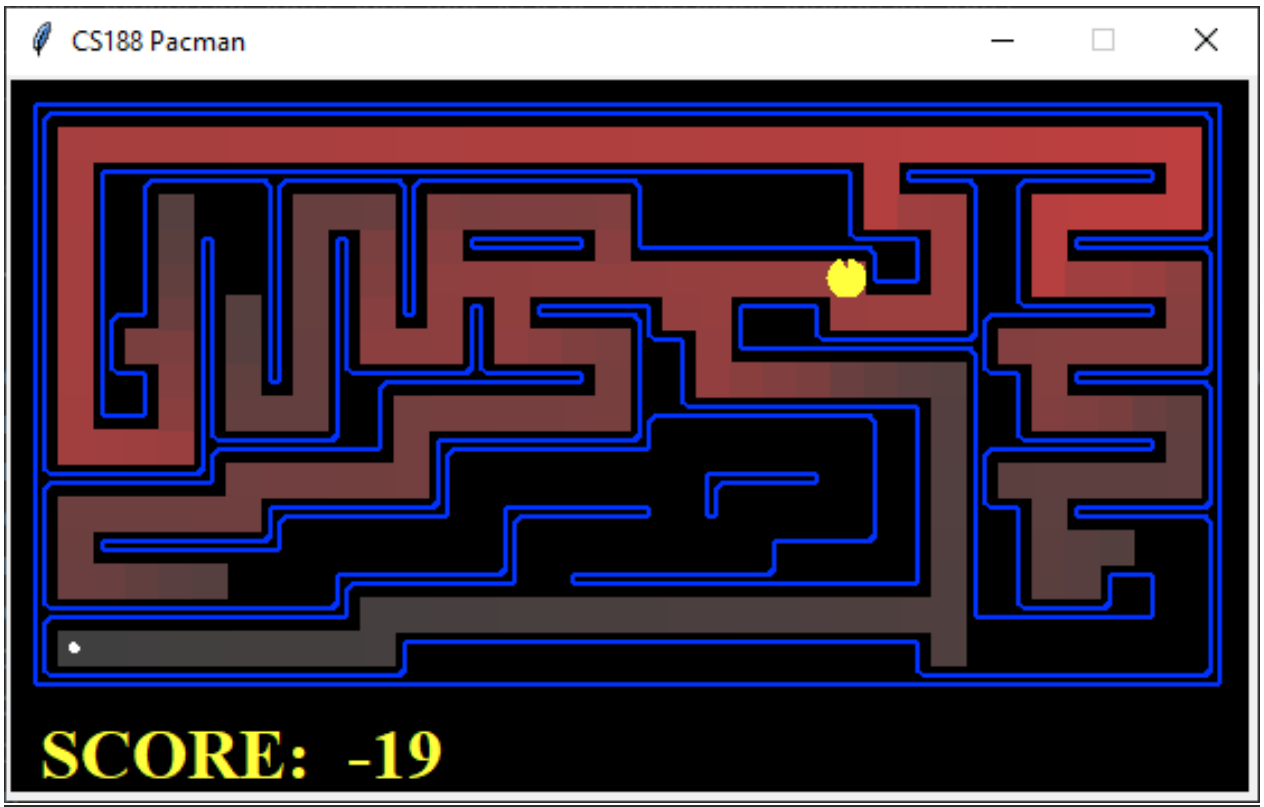


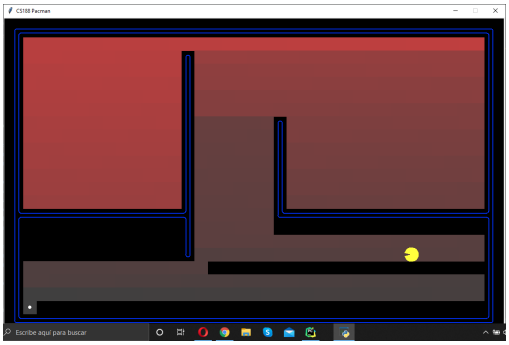
Figura 4: Resultados de la búsqueda A*

5. A* con eliminación de estados repetidos y con una heurística consistente es óptimo. Por ejem-

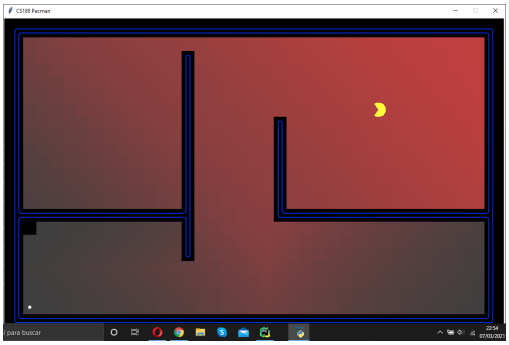
plo, para la configuración `mediumMaze` y con la heurística basada en la distancia Manhattan de pacman al punto, A* encuentra la solución óptima (y también la encuentra en los demás casos de prueba) de coste 68, pero expandiendo 221 nodos, lo que supone una mejora del 21 % respecto a la búsqueda primero en anchura o de coste uniforme.

6. **¿Qué sucede en `openMaze` para las diversas estrategias de búsqueda?**

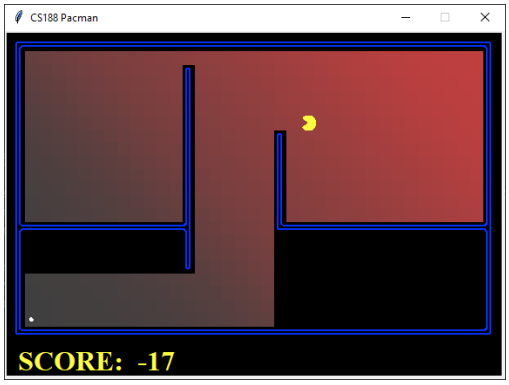
   Este laberinto ejemplifica claramente los fenómenos observados hasta ahora, ya que al no tener apenas paredes se observa mejor cual es la mejor forma de llegar a la comida.



(a) Resultado *DFS*



(b) Resultado *BFS* y *UCS*



(c) Resultado A*

Figura 5: Resultados de distintos algoritmos de búsqueda sobre `openMaze`

Al ejecutar el algoritmo de búsqueda primero en profundidad en este caso, se expande una cantidad razonable de nodos (576), pero el camino resultante está muy lejos de ser óptimo. La longitud de éste último es de 298, muy superior a la longitud de las soluciones encontradas por otros algoritmos.

En el caso de la búsqueda primero en anchura se halla una solución óptima de coste 54, demostrando así la mala calidad del camino hallado por profundidad. Por contra, la eficiencia es bastante peor, ya que se expanden un total de 682 nodos para reallizar este cálculo. Dado que todas las acciones son de coste 1 para este problema, la búsqueda de coste uniforme produce un resultado idéntico.

La aplicación del algoritmo A* (utilizando la distancia de Manhattan como heurística) al laberinto muestra un resultado claramente mejor. Como ya se ha discutido previamente, este

algoritmo es también óptimo, por lo que produce un camino de coste 54. La diferencia frente al resto es que este expande tan solo 535 nodos, lo que supone una clara mejora en cuanto a eficiencia.

Sección 5

1. La solución a este ejercicio se basa en la clase `PositionSerarchProblem`, cuyo código ya estaba implementado. La idea principal de la nueva clase `CornersProblem` es codificar el estado de forma que, además de la posición actual, almacene cuatro variables booleanas que determinen si las esquinas han sido o no recorridas. Estos valores, naturalmente, son inicializados a `False` y posteriormente actualizados cada vez que se alcance una esquina. La función `isGoalState` devolverá `True` solo cuando las cuatro esquinas hayan sido recorridas.
2. Dado que esta implementación es bastante simple, sólo es necesario recurrir a la función `directionToVector` para los cálculos de sucesores.
3.

```
class CornersProblem(search.SearchProblem):

    def __init__(self, startingGameState):
        x, y = self.startingPosition
        self.startState = (x, y, False, False, False, False)  # (x, y, corner[0]
            reached, ... ,corner[3] reached)

    def getStartState(self):
        return self.startState

    def isGoalState(self, state):
        for i in range(4):
            if not state[i + 2]:
                return False
        return True

    def getSuccessors(self, state):
        successors = []
        for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST,
            Directions.WEST]:
            x, y, c0, c1, c2, c3 = state
            c = [c0, c1, c2, c3]
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            if not self.walls[nextx][nexty]:
                for i in range(4):
                    if (nextx, nexty) == self.corners[i]:
                        c[i] = True

                nextc0, nextc1, nextc2, nextc3 = c
                nextState = (nextx, nexty, nextc0, nextc1, nextc2, nextc3)
                successors.append((nextState, action, 1))

        self._expanded += 1 # DO NOT CHANGE
        return successors

    def getCostOfActions(self, actions):
```

```
        if actions == None: return 999999
        x,y= self.startingPosition
        for action in actions:
            dx, dy = Actions.directionToVector(action)
            x, y = int(x + dx), int(y + dy)
            if self.walls[x][y]: return 999999
        return len(actions)
```

4. En la sigiuente captura se muestra la salida del problema de búsqueda de esquinas, que demuestra que se halla una solución, pese a necesitarse 1966 expansiones.

```
C:\Users\leand\Documents\GIT\IA_2021\P1>python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 106 in 0.0 seconds
Search nodes expanded: 1966
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Figura 6: Salida del problema de búsqueda de esquinas

5. Una vez el nuevo problema ha sido creado, se le ha aplicado el algoritmo *BFS*. La búsqueda será completa y, además, óptima si el coste es no decreciente en función de la profundidad (como es el caso de `mediumCorners`). La eficiencia, sim embargo deja bastante que desear, ya que se expanden casi 2000 nodos para este laberinto.

Sección 6

1. Principalmente se tomaron como referencia dos principios para hallar una heurística admisible y consistente. En primer lugar, considerar la distancia Manhattan como métrica, ya que el coste de las acciones (y en consecuencia de las trayectorias) es precisamente la distancia Manhattan recorrida por pacman. En segundo lugar, considerar el problema salvo alguna restricción que simplifique encontrar una cota inferior al coste real. Específicamente, se consideró que un laberinto sin paredes como versión relajada del problema.

2. Para esta sección la única función del framework utilizada (al menos de forma explícita) es `util.manhattanDistance()` para mejorar la legibilidad del código principalmente, pues su utilidad es fácilmente implementable.

3.

```
def cornersHeuristic(state, problem):
    corners = problem.corners # These are the corner coordinates
    x, y, *_ = state
    notvisited = [corners[i] for i in range(4) if not state[i + 2]]
    current = (x, y)
    h = 0
    while len(notvisited) > 0:
        minim = 999999
```

```
            nextCorner = None
            for corner in notvisited:
                d = util.manhattanDistance(current, corner)
                if d < minim:
                    minim = d
                    nextCorner = corner
            h += minim
            current = nextCorner
            notvisited = [corner for corner in notvisited if corner != nextCorner]

        return h
```

4. Aquí se muestra como al añadir la heurística creada se logra hallar la solución con menos de 700 nodos expandidos, frente a los casi 2000 del ejercicio 5.

```
C:\Users\leand\Documents\GIT\IA_2021\P1>python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
Path found with total cost of 106 in 0.0 seconds
Search nodes expanded: 692
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Figura 7: Salida del problema de búsqueda de esquinas aplicando heurística

5. Como se discutirá más adelante, la heurística utilizada es consistente, por tanto el problema de búsqueda en grafo con el algoritmo A* es completo y óptimo. Para `mediumCorners`, pacman encuentra la solución óptima expandiendo 692 nodos.

6. **Explica la lógica de tu heurística.**

   La heurística propuesta es, en síntesis, el coste (o la distancia Manhattan recorrida) del camino más corto que lleva a pacman desde su posición actual a través de las esquinas no visitadas en ese momento considerando que puede atravesar paredes. La manera más intuitiva, o ingenua, de computar esta heurística consiste en sumar la distancia de pacman a la esquina no visitada más cercana, la distancia de esta esquina a su esquina no visitada más cercana y así sucesivamente hasta alcanzar la última de las esquinas no visitadas. Este algoritmo no aplica para hallar el camino más corto entre puntos arbitrarios del plano[1], pero se puede comprobar que funciona como heurística para visitar las esquinas del laberinto.

   Cabe destacar que es fácil demostrar que cuando quedan 1, 2 o 4 esquinas por visitar el camino dado por este procedimiento es el camino más corto. Cuando quedan 3 esquinas, el algoritmo computa el camino más corto si y solo si la esquina más próxima a pacman es adyacente a la esquina ya visitada, es decir, cuando la esquina más cercana a pacman es la opuesta a la visitada, O, el camino más corto no es el que visita primero O sino el que pasa por la segunda esquina más cercana y en su camino a la última esquina pasa por O sin coste adicional. Sin embargo, tras discutirlo con el profesor, se concluye que este último caso no ocurre ya que al

[1] El problema del viajante (TSP) es bastante más complejo.

encontrar la primera esquina, la más cercana jamás es la opuesta. Por tanto, el procedimiento planteado devuelve el camino mínimo en este problema particular.

A partir de lo anterior, esta heurística es admisible y además consistente. Basta demostrar lo segundo:

Demostración. Sea h la heurística propuesta. Para un estado del problema n en el que quedan $k \leq 4$ esquinas por visitar,

$$h(n) := d_1(p, e_1) + d_1(e_1, e_2) + \ldots + d_1(e_{k-1}, e_k) \tag{1}$$

donde p es la posición de pacman en este estado y $e_1, \ldots, e_k$ son las esquinas por visitar en el orden de visita dado por el camino más corto considerando que no hay paredes. Sea n' un sucesor de n, p' la posición de pacman en dicho estado y $e'_1, \ldots, e'_k$ las k esquinas en el orden en el que son atravesadas[2]. Usando que $h(n)$ es el camino más corto de los buscados y la desigualdad triangular de d_1, se tiene:

$$h(n) = d_1(p, e_1) + \sum_{i=1}^{k-1} d_1(e_i, e_{i+1}) \leq d_1(p, e'_1) + \sum_{i=1}^{k-1} d_1(e'_i, e'_{i+1}) \leq \tag{2}$$

$$\leq d_1(p, p') + d_1(p', e'_1) + \sum_{i=1}^{k-1} d_1(e'_i, e'_{i+1}) = coste(n, n') + h(n') \tag{3}$$

□

Sección 7

En síntesis, la realización de esta práctica permite aplicar y experimentar con los conceptos y resultados estudiados en la parte teórica de la asignatura, y conduce a las siguientes conclusiones:

- Los algoritmos de búsqueda, al menos los estudiados, consisten de los mismos pasos y difieren únicamente en el orden de expansión de los nodos generados, lo que se identifica con el uso de distintas estructuras de datos en las que almacenar la lista de nodos abiertos. Además, la diferencia entre los algoritmos ciegos y los informados es que los segundos incluyen una heurística a la hora de decidir el siguiente nodo a expandir.
- La formalización de un problema de búsqueda, en general, no es trivial y una buena formalización ayuda a encontrar soluciones más fácilmente.
- La búsqueda A* con una heurística consistente y cercana al coste real del camino menos costoso a un estado final supone una importante mejora respecto a los algoritmos no informados. Por ejemplo, para el problema de hallar las esquinas, A* es un 184 % mejor que la búsqueda primero en anchura en cuanto al número de nodos expandidos.

[2]Como n' solo está un paso por delante de n, en el caso en que p' sea la ubicación de una esquina, se puede considerar que en ese paso sigue habiendo k esquinas por visitar, solo que la distancia de pacman a la esquina más cercana es cero.