BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

Técnicas de Inteligencia Artificial

Práctica 1 Pac-Man

2023-2024

...

Titulación:

Grado en Informática de Gestión y Sistemas de Información

...

4º Curso (1º Cuatrimestre)

...

Página de GitHub

Alan García, Álvaro Díez-Andino

15 de octubre de 2023

Índice

1. Pregunta1: Depth First Search

1.1. Descripción

El Depth First Search (DFS), o busqueda en profundidad es un algoritmo de busqueda **no informada**. Eso quiere decir que es un algoritmo que no emplea ninguna información adicional que no sean los nodos y sus arcos. Si tuviesemos información sobre la meta, por ejemplo su posición, o alguna otra medida que nos indique si vamos bien o mal encaminados en nuestra busqueda, podríamos emplearla para guiarnos en la busqueda y entonces estaríamos hablando de una busqueda **informada**. El DFS es una busqueda muy primitiva a ese respecto. Simplemente introduce una sistematizidad, una estrategia en la busqueda; sigue buscando por el último nodo que has expandido. Garantiza encontrar la meta, pero no garantiza encontrar el camino óptimo. No es eficiente porque en el peor de los casos tiene que explorar todos los nodos hasta encontrar la meta.

Solamente necesita:

- una función que te indique si el nodo o estado actual es la meta o no (*isGoalState()*)
- una función de transición que te devuelve una estructura de datos con los estado a los que puedes transitar desde el actual y la acción asociada, es decir, qué nodos se pueden expandir a partir del actual (*getSucessors()*).
- la función que devuelve el estado o nodo *meta*

Es importante recalcar:

- Necesita una estructura de datos (frontera o fringe) para almacenar los nodos que han sido expandidos pero aún no explorados. Se empleará una *pila* dado que queremos que el siguiente nodo a expandir sea último nodo añadido para ser explorado.
- hace falta almacenar junto con cada nodo el camino desde el comienzo hasta el nodo.
- Hace falta una estructura para almacenar los nodos ya explorados para evitar ciclos

1.1.1. Tipo de Algoritmo

- Algoritmo de busqueda **no informado**

1.2. Algoritmo

DFS(u)

MIENTRAS haya elementos en la frontera (elementos a expandir)

nodoAct = obtener el siguiente nodo de la frontera

SI nodoAct no ha sido visitado:

SI nodoAct es meta:

devolver el camino

- SINO:

FOR $(nodoSuc, dir) \in sucesores$ DO

introducir nodoSuc en la frontera

1.2.1. Código V1

```python
def depthFirstSearch(problem):
    """
    Search the deepest nodes in the search tree first.

    Your search algorithm needs to return a list of actions that reaches the
    goal. Make sure to implement a graph search algorithm.

    To get started, you might want to try some of these simple commands to
    understand the search problem that is being passed in:

    print("Start:", problem.getStartState())
    print("Is the start a goal?", problem.isGoalState(problem.getStartState()))
    print("Start's successors:", problem.getSuccessors(problem.getStartState()))
    """
    "*** YOUR CODE HERE ***"

    print("---- Depth First Search")
    print(problem.getStartState())
    print(problem.getSuccessors(problem.getStartState()))
    print(problem.getSuccessors((4, 5)))
    print(problem.isGoalState((1,1)))

    porVisitar = [(problem.getStartState(), [])] #Stack -> Inicializado con la posición inicial y la lista de movimientos hasta el estado
    visitados = set() # Set
    nodoFinal = None

    while nodoFinal is None and len(porVisitar) != 0:
        nodoAct = porVisitar.pop() # Visitamos el siguiente nodo

        if nodoAct[0] not in visitados:
            # No habiamos visitado anteriormente este nodo y registramos la posición actual como visitada
            visitados.add(nodoAct[0])

            if problem.isGoalState(nodoAct[0]):
                # Hemos encontrado el camino
                nodoFinal = nodoAct
            else:
                # Aun pueden quedar estados por expandir
                sucesores = problem.getSuccessors(nodoAct[0])

                for s in sucesores:
                    # Copiamos el camino del padre y añadimos la acción para llegar al nuevo estado
                    camino = nodoAct[1].copy()
                    camino.append(s[1])
                    pos = s[0]
                    porVisitar.append((pos, camino))

    if nodoFinal is None:
        # No hemos encontrado un camino que lleve al objetivo
        return []

    # Devolvemos el camino hasta el objetivo
    return nodoFinal[1]
```

```
Finished at 18:34:59

Provisional grades
==================
Question q1: 3/3
Question q2: 3/3
Question q3: 0/3
Question q4: 0/3
Question q5: 0/3
Question q6: 0/3
Question q7: 0/4
Question q8: 0/3
------------------
Total: 6/25

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.

ag6154lk@msidealan:~/IngInfor/TIA/Labo_1eGela$
```

Figura 1: Primea versión DFS

1.2.2. Código V-Final

```python

def depthFirstSearch(problem):
    """
    Search the deepest nodes in the search tree first.

    Your search algorithm needs to return a list of actions that reaches the
    goal. Make sure to implement a graph search algorithm.

    To get started, you might want to try some of these simple commands to
    understand the search problem that is being passed in:

    print("Start:", problem.getStartState())
    print("Is the start a goal?", problem.isGoalState(problem.getStartState()))
    print("Start's successors:", problem.getSuccessors(problem.getStartState()))
    """
    "*** YOUR CODE HERE ***"

    print("---- Depth First Search")
    #print(problem.getStartState())
    #print(problem.getSuccessors(problem.getStartState()))
    #print(problem.getSuccessors((4, 5)))
    #print(problem.isGoalState((1,1)))
    porVisitar = [(problem.getStartState(), [])]
    #porVisitar = util.Stack([(problem.getStartState(), [])]) #Stack -> Inicializado con la posición inicial y la lista de movimientos hasta el estado
    visitados = set() # Set
    nodoFinal = None

    while nodoFinal is None and len(porVisitar) != 0:
        nodoAct = porVisitar.pop() # Visitamos el siguiente nodo

        if nodoAct[0] not in visitados:
            # No habiamos visitado anteriormente este nodo y registramos la posición actual como visitada
            visitados.add(nodoAct[0])

            if problem.isGoalState(nodoAct[0]):
                # Hemos encontrado el camino
                nodoFinal = nodoAct
            else:
                # Aun pueden quedar estados por expandir
                sucesores = problem.getSuccessors(nodoAct[0])

                for s in sucesores:
                    # Copiamos el camino del padre y añadimos la acción para llegar al nuevo estado
                    camino = nodoAct[1].copy()
                    camino.append(s[1])
                    pos = s[0]
                    porVisitar.append((pos, camino))

    if nodoFinal is None:
        # No hemos encontrado un camino que lleve al objetivo
        return []

    # Devolvemos el camino hasta el objetivo
    return nodoFinal[1]
```

```
Finished at 18:40:56

Provisional grades
==================
Question q1: 3/3
Question q2: 0/3
Question q3: 0/3
Question q4: 0/3
Question q5: 0/3
Question q6: 0/3
Question q7: 0/4
Question q8: 0/3
------------------
Total: 3/25

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.

ag6154lk@msidealan:~/IngInfor/TIA/Labo_1eGela$
```

Figura 2: Versión final DFS

1.2.3. Problemas y dificultades encarados en el ejercicio

En la primera versión utilicé una lista simple de python en lugar de la clase *Stack* proporcionada en la librería *util*, aunque el funcionamiento era el mismo.

La mayor dificultad que me encontré realizando este ejercicio fue el retorno del camino. Al final, he conseguido devolverlo guardando el camino seguido para llegar a cada uno de los nodos.

1.2.4. Ejemplo

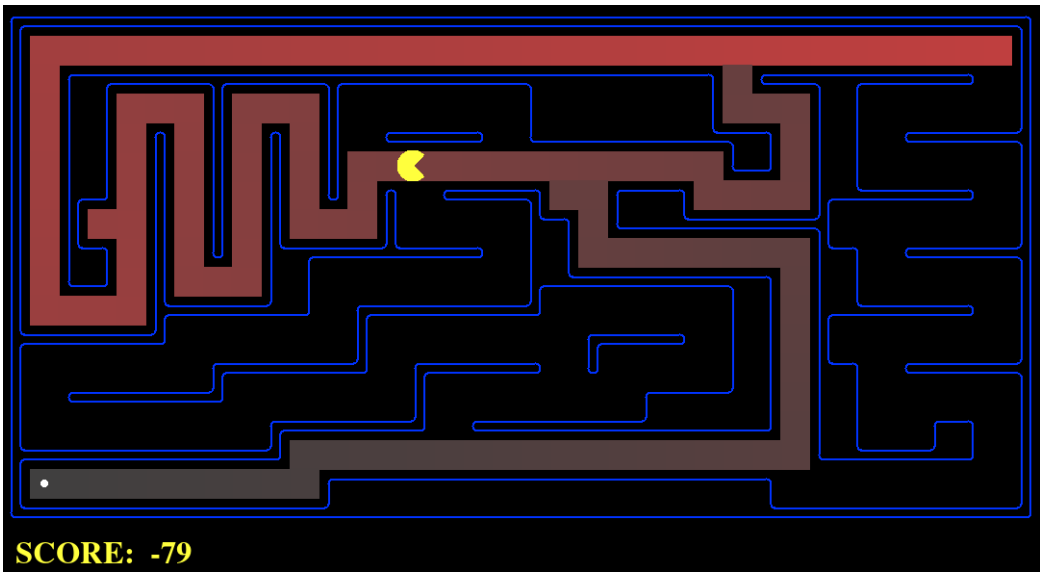

Figura 3: Ejemplo DFS

2. Pregunta2: Breadth First Search

2.1. Descripción

El algoritmo Breadth-First Search (Búsqueda en Amplitud) es un algoritmo utilizado para recorrer o buscar en estructuras de datos como grafos o árboles. Este tipo de algoritmo de búsqueda no informado comienza por el nodo raíz y se mueve gradualmente a través de los nodos vecinos a una profundidad dada antes de avanzar a los nodos vecinos de la siguiente profundidad. Este enfoque garantiza que primero se visiten todos los nodos a una profundidad dada antes de descender a una profundidad mayor. El algoritmo BFS garantiza que se visiten todos los nodos alcanzables desde el nodo raíz,

pero no necesariamente en el orden más eficiente en términos de tiempo o memoria.

Este algoritmo tiene los mismos requerimientos que el DFS, siendo la única diferencia la estructura de datos utilizada para almacenar el fringe o frontera (se emplea una cola en lugar de una pila).

2.1.1. Tipo de Algoritmo

- Algoritmo de busqueda **no informado**

2.2. Algoritmo

BFS(u)

MIENTRAS haya elementos en la frontera (elementos a expandir)

nodoAct = obtener el siguiente nodo de la frontera

SI nodoAct no ha sido visitado:

SI nodoAct es meta:

devolver el camino

- SINO:

FOR $(nodoSuc, dir) \in sucesores$ DO

introducir nodoSuc en la frontera

2.2.1. Código V1

```python

def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    # Es igual que el DFS pero en lugar de utilizar un stack usamos una cola
    # Solo cambia el nodo que miramos primero
    porVisitar = util.Queue([(problem.getStartState(), [])]) #Stack -> Inicializado con la posición inicial y la lista de movimientos hasta el estado
    visitados = set() # Set
    nodoFinal = None

    while nodoFinal is None and not porVisitar.isEmpty():
        nodoAct = porVisitar.pop() # Visitamos el siguiente nodo

        if nodoAct[0] not in visitados:
            # No habiamos visitado anteriormente este nodo y registramos la posición actual como visitada
            visitados.add(nodoAct[0])

            if problem.isGoalState(nodoAct[0]):
                # Hemos encontrado el camino
                nodoFinal = nodoAct
            else:
                # Aun pueden quedar estados por expandir
                sucesores = problem.getSuccessors(nodoAct[0])

                for s in sucesores:
                    # Copiamos el camino del padre y añadimos la acción para llegar al nuevo estado
                    camino = nodoAct[1].copy()
                    camino.append(s[1])
                    pos = s[0]
                    porVisitar.push((pos, camino))

    if nodoFinal is None:
        # No hemos encontrado un camino que lleve al objetivo
        return []

    # Devolvemos el camino hasta el objetivo
    return nodoFinal[1]

```

```
Finished at 18:34:59

Provisional grades
==================
Question q1: 3/3
Question q2: 3/3
Question q3: 0/3
Question q4: 0/3
Question q5: 0/3
Question q6: 0/3
Question q7: 0/4
Question q8: 0/3
------------------
Total: 6/25

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.

ag6154lk@msidealan:~/IngInfor/TIA/Labo_1eGela$
```

Figura 4: Primea versión BFS

2.2.2. Código V-Final

```python

def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    # Es igual que el DFS pero en lugar de utilizar un stack usamos una cola
    # Solo cambia el nodo que miramos primero
    porVisitar = util.Queue([(problem.getStartState(), [])]) #Stack -> Inicializado con la posición inicial y la lista de movimientos hasta el estado
    visitados = set() # Set
    nodoFinal = None

    while nodoFinal is None and not porVisitar.isEmpty():
        nodoAct = porVisitar.pop() # Visitamos el siguiente nodo

        if nodoAct[0] not in visitados:
            # No habiamos visitado anteriormente este nodo y registramos la posición actual como visitada
            visitados.add(nodoAct[0])

            if problem.isGoalState(nodoAct[0]):
                # Hemos encontrado el camino
                nodoFinal = nodoAct
            else:
                # Aun pueden quedar estados por expandir
                sucesores = problem.getSuccessors(nodoAct[0])

                for s in sucesores:
                    # Copiamos el camino del padre y añadimos la acción para llegar al nuevo estado
                    camino = nodoAct[1].copy()
                    camino.append(s[1])
                    pos = s[0]
                    porVisitar.push((pos, camino))

    if nodoFinal is None:
        # No hemos encontrado un camino que lleve al objetivo
        return []

    # Devolvemos el camino hasta el objetivo
    return nodoFinal[1]

```

```
Finished at 18:34:59

Provisional grades
==================
Question q1: 3/3
Question q2: 3/3
Question q3: 0/3
Question q4: 0/3
Question q5: 0/3
Question q6: 0/3
Question q7: 0/4
Question q8: 0/3
------------------
Total: 6/25

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.

ag6154lk@msidealan:~/IngInfor/TIA/Labo_1eGela$
```

Figura 5: Versión final BFS

2.2.3. Problemas y dificultades encarados en el ejercicio

En este ejercicio no he encontrado ninguna dificultad. Simplemente he reemplazado la estructura de datos del *fringe* utilizada en el *Depth First Search* por una cola en lugar de una pila.

2.2.4. Ejemplo

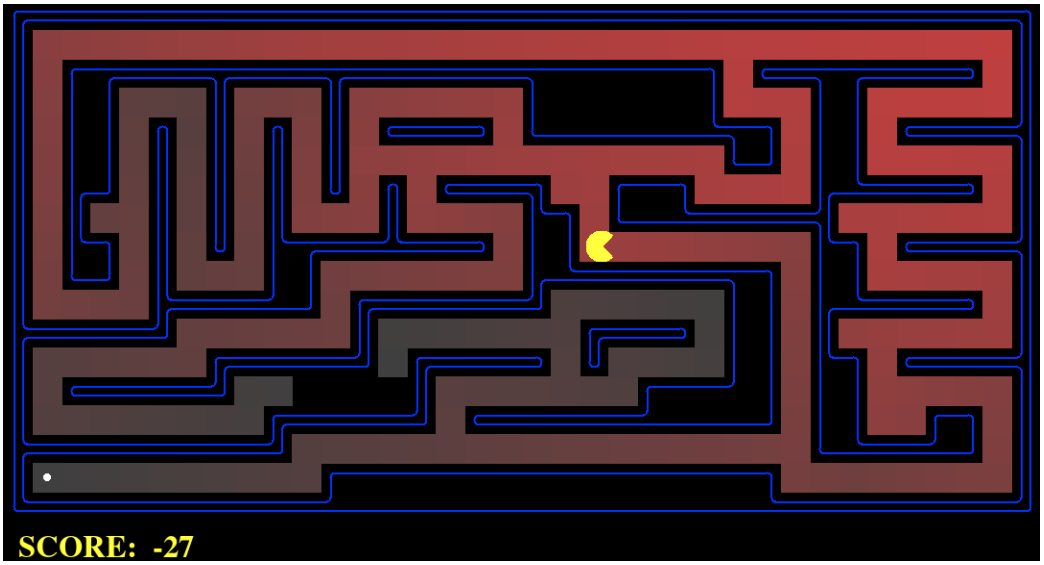

Figura 6: Ejemplo BFS

3. Pregunta3: Uniform Cost Search

3.1. Descripción

El Uniform Cost Search (UCS), también conocido como Búsqueda de Costo Uniforme, es un algoritmo de búsqueda de caminos en grafos que se utiliza para encontrar la ruta más corta desde un nodo de inicio hacia un nodo de destino en un grafo ponderado. se guía por el principio de expandir primero los nodos de menor costo, lo que significa que examina los nodos que tienen el menor costo acumulado desde el nodo de inicio hasta ese punto. Para lograr esto, el UCS mantiene una lista de nodos abiertos (open set) y utiliza una estructura de datos como una cola de prioridad o una lista ordenada para asegurarse de que el nodo con el menor costo acumulado se expanda primero.

Solamente necesita:

- Una función que te indique si el nodo o estado actual es la meta o no
- Una función de transición que te devuelve una estructura de datos con los estado a los que puedes transitar desde el actual y la acción asociada, es decir, qué nodos se pueden expandir a partir del actual (*getSucessors()*).

Es importante recalcar:

- Necesita una estructura de datos (frontera o fringe) para almacenar los nodos que han sido expandidos pero aún no explorados. Se empleará una *Cola de prioridad* dado que queremos que el siguiente nodo a expandir sea el de menor costo acumulado.
- Hace falta almacenar junto con cada nodo el camino desde el comienzo hasta el nodo.
- Hace falta una estructura para almacenar los nodos ya explorados para evitar ciclos.

3.1.1. Tipo de Algoritmo

- Algoritmo de busqueda **no informado**

3.2. Algoritmo

UCS(u)

MIENTRAS haya elementos en la frontera (elementos a expandir)

nodoAct = obtener el siguiente nodo de la frontera

SI nodoAct no ha sido visitado:

SI nodoAct es meta:

devolver el camino

- SINO:

FOR $(nodoSuc, dir) \in sucesores$ DO

introducir nodoSuc en la frontera con el costo total de costoAct + costoSuc

3.2.1. Código V1

```python

def uniformCostSearch(problem):
    """Search the node of least total cost first."""
    "*** YOUR CODE HERE ***"
    porVisitar = util.PriorityQueue() #Stack -> Inicializado con la posición inicial y la lista de movimientos hasta el estado
    porVisitar.push((problem.getStartState(), [], 0), 0)
    visitados = set() # Set
    nodoFinal = None

    while nodoFinal is None and not porVisitar.isEmpty():
        nodoAct = porVisitar.pop() # Visitamos el siguiente nodo

        if nodoAct[0] not in visitados:
            # No habiamos visitado anteriormente este nodo y registramos la posición actual como visitada
            visitados.add(nodoAct[0])

            if problem.isGoalState(nodoAct[0]):
                # Hemos encontrado el camino
                nodoFinal = nodoAct
            else:
                # Aun pueden quedar estados por expandir
                sucesores = problem.getSuccessors(nodoAct[0])

                for s in sucesores:
                    # Copiamos el camino del padre y añadimos la acción para llegar al nuevo estado
                    pos = s[0]
                    camino = nodoAct[1].copy()
                    camino.append(s[1])
                    coste = nodoAct[2] + s[2]
                    porVisitar.push((pos, camino, coste), coste)

    if nodoFinal is None:
        # No hemos encontrado un camino que lleve al objetivo
        return []

    # Devolvemos el camino hasta el objetivo
    return nodoFinal[1]

```

```
Provisional grades
==================
Question q1: 3/3
Question q2: 3/3
Question q3: 3/3
Question q4: 0/3
Question q5: 0/3
Question q6: 0/3
Question q7: 0/4
Question q8: 0/3
------------------
Total: 9/25

Your grades are NOT yet registered.  To register yo
to follow your instructor's guidelines to receive c

ag6154lk@msidealan:~/IngInfor/TIA/Labo_1eGela$
```

Figura 7: Primea versión UCS

3.2.2. Código V-Final

Es el mismo código y resultados que en la primera versión.

3.2.3. Problemas y dificultades encarados en el ejercicio

Al igual que en el ejercicio anterior, el único cambio que he realizado en este algoritmo ha sido la estructura de datos que define los nodos *porVisitar* (nodos que establecen el borde o fringe).

3.2.4. Ejemplo

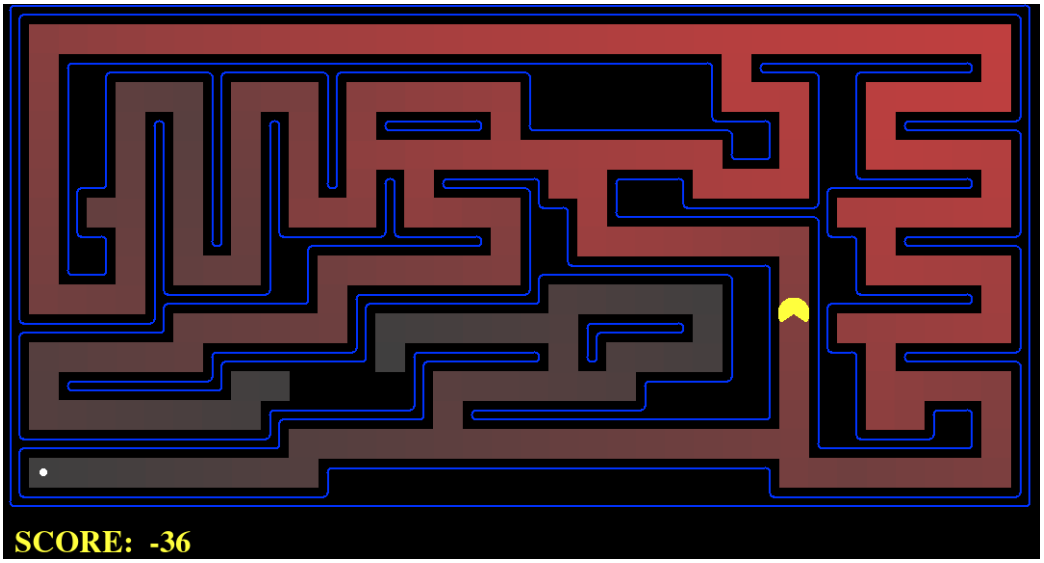

Figura 8: Ejemplo UCS

4. Pregunta4: A* Search

4.1. Descripción

El algoritmo A* es un algoritmo de búsqueda heurística que se utiliza para encontrar el camino más corto desde un nodo inicial a un nodo objetivo en un grafo o un espacio de búsqueda. El algoritmo A* combina las ventajas de la búsqueda en amplitud (BFS) y la búsqueda en profundidad (DFS) al considerar tanto el costo acumulado actual como una estimación heurística del costo futuro para cada nodo.

Solamente necesita:

- Una función que te indique si el nodo o estado actual es la meta o no
- Una función de transición que te devuelve una estructura de datos con los estado a los que puedes transitar desde el actual y la acción asociada, es decir, qué nodos se pueden expandir a partir del actual (*getSucessors()*).
- Un heurístico que indique el coste de cada estado

Es importante recalcar:

- Necesita una estructura de datos (frontera o fringe) para almacenar los nodos que han sido expandidos pero aún no explorados. Se empleará una *Cola de prioridad* dado que queremos que el siguiente nodo a expandir sea el de menor costo acumulado.
- Hace falta almacenar junto con cada nodo el camino desde el comienzo hasta el nodo.
- Hace falta una estructura para almacenar los nodos ya explorados para evitar ciclos.

4.2. Algoritmo

A*STAR(u)

Mientras nodoFinal sea nulo y porVisitar no esté vacía

Tomar el siguiente nodo de la cola porVisitar y asignarlo a nodoAct

Obtener el estado desde nodoAct

Si el estado no está en la lista de visitados entonces

Agregar el estado a la lista de visitados

Obtener el costo acumulado del nodo actual y asignarlo a costeNodoAct

Si el estado es el objetivo entonces

Asignar nodoAct a nodoFinal

Si no

Obtener los sucesores del estado

Para cada sucesor en la lista de sucesores hacer

Obtener la nueva posición y copiar el camino desde el padre

Agregar la acción del sucesor al camino

Calcular el nuevo costo como el costo acumulado más el costo del sucesor

Calcular la prioridad como el nuevo costo más el valor heurístico del nuevo estado

Agregar el estado, el camino y la prioridad a la cola porVisitar

return (listaMovimientos)

4.2.1. Tipo de Algoritmo

- Algoritmo de busqueda **informado**

4.2.2. Código V1

```python
def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""
    "*** YOUR CODE HERE ***"

    # INICIALIZACIÓN
    porVisitar = util.PriorityQueue() #Cola de prioridad -> Inicializado con la posición inicial y la lista de movimientos hasta el estado
    posInicial = problem.getStartState()
    costeInicial = heuristic(posInicial, problem)
    porVisitar.push((posInicial, [], costeInicial), costeInicial)
    visitados = set() # Set
    nodoFinal = None

    # ALGORITMO
    while nodoFinal is None and not porVisitar.isEmpty():
        nodoAct = porVisitar.pop() # Visitamos el siguiente nodo

        if nodoAct[0] not in visitados:
            # No habiamos visitado anteriormente este nodo y registramos la posición actual como visitada
            visitados.add(nodoAct[0])
            # El coste de llegar hasta el nodo actual es el coste acumulado menos el heurístico del nodo actual.
            # Antes se sumó el heuristico, por lo que para saber el coste hay que restarlo
            costeNodoAct = nodoAct[2] - heuristic(nodoAct[0], problem)

            if problem.isGoalState(nodoAct[0]):
                # Hemos encontrado el camino
                nodoFinal = nodoAct
            else:
                # Aun pueden quedar estados por expandir
                sucesores = problem.getSuccessors(nodoAct[0])

                for s in sucesores:
                    # Copiamos el camino del padre y añadimos la acción para llegar al nuevo estado
                    pos = s[0]
                    camino = nodoAct[1].copy()
                    camino.append(s[1])

                    # El coste del sucesor va a ser el coste acumulado hasta el más el heurístico del sucesor
                    coste =  costeNodoAct + s[2] + heuristic(pos, problem)
                    porVisitar.push((pos, camino, coste), coste)

    # RETURN
    if nodoFinal is None:
        # No hemos encontrado un camino que lleve al objetivo
        return []

    # Devolvemos el camino hasta el objetivo
    return nodoFinal[1]
```

```
Provisional grades
==================
Question q1: 3/3
Question q2: 3/3
Question q3: 3/3
Question q4: 3/3
Question q5: 0/3
Question q6: 0/3
Question q7: 0/4
Question q8: 0/3
------------------
Total: 12/25

Your grades are NOT yet registered.  To regist
to follow your instructor's guidelines to rece

ag6154lk@msidealan:~/IngInfor/TIA/Labo_1eGela$
```

Figura 9: Primera versión AStar

4.2.3. Código V-Final

```python
def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""
    "*** YOUR CODE HERE ***"

    # INICIALIZACIÓN
    porVisitar = util.PriorityQueue() #Cola de prioridad -> Inicializado con la posición inicial y la lista de movimientos hasta el estado
    posInicial = problem.getStartState()
    costeInicial = heuristic(posInicial, problem)
    porVisitar.push((posInicial, [], costeInicial), costeInicial)
    visitados = set() # Set
    nodoFinal = None

    # ALGORITMO
    while nodoFinal is None and not porVisitar.isEmpty():
        nodoAct = porVisitar.pop() # Visitamos el siguiente nodo

        if nodoAct[0] not in visitados:
            # No habiamos visitado anteriormente este nodo y registramos la posición actual como visitada
            visitados.add(nodoAct[0])
            costeNodoAct = nodoAct[2]

            if problem.isGoalState(nodoAct[0]):
                # Hemos encontrado el camino
                nodoFinal = nodoAct
            else:
                # Aun pueden quedar estados por expandir
                sucesores = problem.getSuccessors(nodoAct[0])

                for s in sucesores:
                    # Copiamos el camino del padre y añadimos la acción para llegar al nuevo estado
                    pos = s[0]
                    camino = nodoAct[1].copy()
                    camino.append(s[1])

                    # El coste del sucesor va a ser el coste acumulado hasta él
                    coste =  costeNodoAct + s[2]
                    # Ordenamos la cola teniendo en cuenta el coste acumulado más el heurístico
                    porVisitar.push((pos, camino, coste), coste + heuristic(pos, problem) )

    # RETURN
    if nodoFinal is None:
        # No hemos encontrado un camino que lleve al objetivo
        return []

    # Devolvemos el camino hasta el objetivo
    return nodoFinal[1]

```

```
Finished at 11:21:49

Provisional grades
==================
Question q1: 3/3
Question q2: 3/3
Question q3: 3/3
Question q4: 3/3
Question q5: 0/3
Question q6: 0/3
Question q7: 0/4
Question q8: 0/3
------------------
Total: 12/25

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.

ag6154lk@msidealan:~/IngInfor/TIA/Labo_1eGela$
```

Figura 10: Versión final AStar

4.2.4. Problemas y dificultades encarados en el ejercicio

En este ejercicio, la mayor dificultad ha estado en pensar cómo relacionar el heurístico y el coste acumulado. Al principio, la solución planteada consistía en establecer el coste de los sucesores como su coste acumulado y el heurístico de cada uno de los sucesores y luego, a la hora de visitar uno de los sucesores previamente añadidos, obteníamos el coste acumulado de la resta del coste del nodo actual menos su heurístico. Sin embargo, una solución más simple es dejar el coste acumulado de cada nodo igual que en el algoritmo UCS y emplear la suma del coste acumulado y el heurístico solo para ordenar la cola de prioridad.

4.2.5. Ejemplo

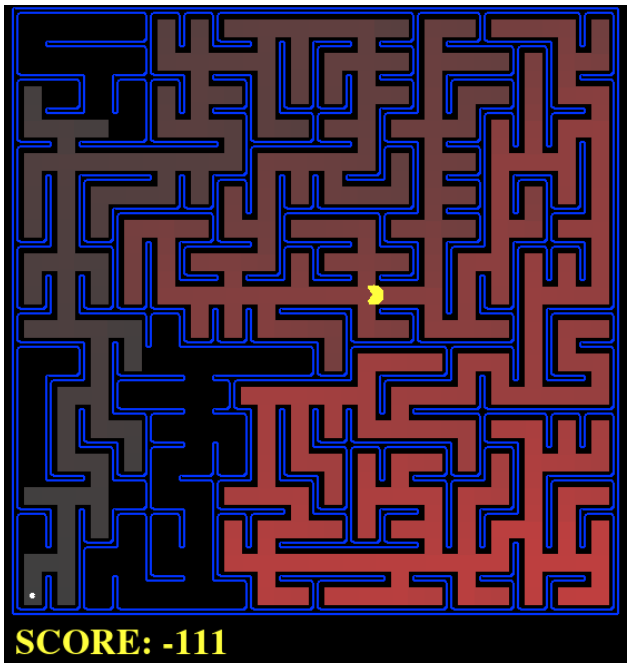

Figura 11: Ejemplo AStar

5. Pregunta5: All the Corners Problem

5.1. Descripción

La estrategia de búsqueda .ªll the Corners"(ATC), también conocida como "Búsqueda en Todas las Esquinas", es un enfoque específico utilizado en

juegos o problemas donde el objetivo es visitar todas las esquinas o áreas específicas de un espacio de búsqueda antes de completar otra tarea.

Solamente necesita:

- Modificación del estado: Para este problema el estado estará compuesto por la posición del Pac-Man y la información sobre las esquinas que haya visitado.
- Modificación del goal: El goal ya no consistirá llegar a un punto X si no que será haber visitado todas las esquinas.
- Modificación del estado inicial: Además de la posición inicial de Pac-Man se deberá buscar las esquinas que se deberán visitar.

5.1.1. Código V1

```
class CornersProblem(search.SearchProblem):
    """
    This search problem finds paths through all four corners of a layout.

    You must select a suitable state space and successor function
    """

    def __init__(self, startingGameState):
        """
        Stores the walls, pacman's starting position and corners.
        """
        self.walls = startingGameState.getWalls()
        self.startingPosition = startingGameState.getPacmanPosition()
        top, right = self.walls.height - 2, self.walls.width - 2
        self.corners = ((1, 1), (1, top), (right, 1), (right, top))
        for corner in self.corners:
            if not startingGameState.hasFood(*corner):
                print('Warning: no food in corner ' + str(corner))
        self._expanded = 0  # DO NOT CHANGE; Number of search nodes expanded
        # Please add any code here which you would like to use
        # in initializing the problem

```

```
23          "*** YOUR CODE HERE ***"
24
25          # El estado est  definido por la poosici n del pac-
    man y por las esquinas que no ha visitado
26          self.startState = (self.startingPosition, self.
    getListaCornersNoVisitados(self.corners, self.
    startingPosition))
27
28      def getListaCornersNoVisitados(self, lista_corners,
    posicion):
29          # Recibe la lista de los corners y la posicion del
    pac-man
30          # Devuelve la lista de los corners no visitados
31          cornersNoVisitados = []
32          for corner in self.corners:
33              if posicion != corner:
34                  cornersNoVisitados.append(corner)
35
36          return cornersNoVisitados
37
38      def getStartState(self):
39          """
40          Returns the start state (in your state space, not the
     full Pacman state
41          space)
42          """
43          "*** YOUR CODE HERE ***"
44          return self.startState
45
46
47      def isGoalState(self, state):
48          """
49          Returns whether this search state is a goal state of
    the problem.
50          """
51          "*** YOUR CODE HERE ***"
52
53          # El pac-man ha visitado todas las esquinas?
54          return len(state[1]) == 0
55
56
57      def getSuccessors(self, state):
58          """
59          Returns successor states, the actions they require,
    and a cost of 1.
60
61           As noted in search.py:
62              For a given state, this should return a list of
    triples, (successor,
```

```
63            action, stepCost), where 'successor' is a
    successor to the current
64            state, 'action' is the action required to get
    there, and 'stepCost'
65            is the incremental cost of expanding to that
    successor
66        """
67        successors = []
68        for action in [Directions.NORTH, Directions.SOUTH,
    Directions.EAST, Directions.WEST]:
69            # Add a successor state to the successor list if
    the action is legal
70            # Here's a code snippet for figuring out whether
    a new position hits a wall:
71            #   x,y = currentPosition
72            #   dx, dy = Actions.directionToVector(action)
73            #   nextx, nexty = int(x + dx), int(y + dy)
74            #   hitsWall = self.walls[nextx][nexty]
75
76            "*** YOUR CODE HERE ***"
77            posicion, cornersNoVisitados = state
78
79            dx, dy = Actions.directionToVector(action)
80            nextx, nexty = int(posicion[0] + dx), int(
    posicion[1] + dy)
81
82            if not self.walls[nextx][nexty]:
83                newPosicion = (nextx, nexty)
84                newCornersNoVisitados = self.
    getListaCornersNoVisitados(cornersNoVisitados, newPosicion
    )
85                newState = (newPosicion,
    newCornersNoVisitados)
86
87                cost = 1 # Coste uniforme provisional
88
89                successors.append((newState, action, cost))
90
91        self._expanded += 1  # DO NOT CHANGE
92        return successors
93
94    def getCostOfActions(self, actions):
95        """
96        Returns the cost of a particular sequence of actions.
      If those actions
97        include an illegal move, return 999999.  This is
    implemented for you.
98        """
99        if actions is None: return 999999
```

```
100         x, y = self.startingPosition
101         for action in actions:
102             dx, dy = Actions.directionToVector(action)
103             x, y = int(x + dx), int(y + dy)
104             if self.walls[x][y]: return 999999
105         return len(actions)
```

Listing 1: Python example

```
Finished at 12:54:31

Provisional grades
==================
Question q1: 3/3
Question q2: 3/3
Question q3: 3/3
Question q4: 3/3
Question q5: 0/3
Question q6: 0/3
Question q7: 0/4
Question q8: 0/3
-----------------
Total: 12/25

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.

ag6154lk@msidealan:~/IngInfor/TIA/Labo_1eGela$
```

Figura 12: Primera versión ATC

5.1.2. Código V-Final

```
308
309     def getListaCornersNoVisitados(self, lista_corners, posicion):
310         # Recibe la lista de los corners y la posicion del pac-man
311         # Devuelve la lista de los corners no visitados
312         cornersNoVisitados = []
313         for corner in lista_corners:
314             if posicion != corner:
315                 cornersNoVisitados.append(corner)
316
317         return cornersNoVisitados
318
```

```
Finished at 13:14:43

Provisional grades
==================
Question q1: 3/3
Question q2: 3/3
Question q3: 3/3
Question q4: 3/3
Question q5: 3/3
Question q6: 0/3
Question q7: 0/4
Question q8: 0/3
------------------
Total: 15/25

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.

ag6154lk@msidealan:~/IngInfor/TIA/Labo_1eGela$
```

Figura 13: Versión final ATC

5.1.3. Problemas y dificultades encarados en el ejercicio

La mayor dificultad la hemos encontrado al inicio puesto que hemos tardado en visualizar el objetivo de este problema. Una vez descubierto el objetivo, hemos pensado varias formas de representar el estado.

- Primera idea: Posición del Pac-Man y una tupla con cada esquina y si ha sido visitada. Esta idea ha sido descartada rápidamente ya que es más compleja de lo que debería ser.
- Segunda idea: Posición del Pac-Man y una lista de esquinas visitadas.
- Tercera idea: Posición del Pac-Man y una lista de esquinas no visitadas, muy parecida a la segunda idea pero nos ha parecido una implementación mas sencilla.

Finalmente se ha decidido por implementar la tercera idea. Durante esta implementación hemos creado dos versiones: en la primera

- Primera versión: El Pac-Man se quedaba parado ya que no podía llegar al goal.
- Segunda idea: Pac-Man llega a todas las esquinas, el error consistía en que para saber si había al goal miraba una lista de esquinas no actualizada y por ende se quedaba en bucle.

5.1.4. Ejemplo

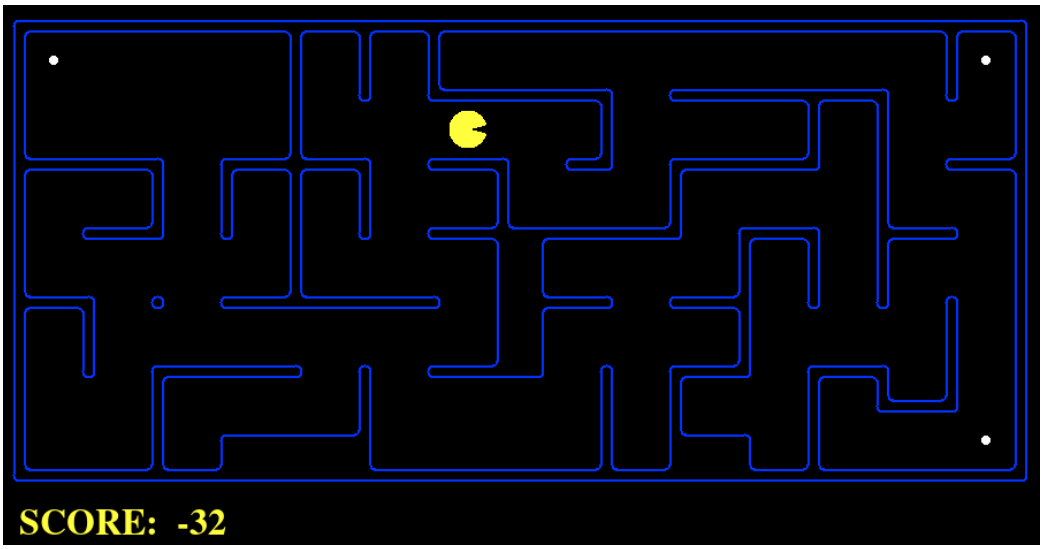

Figura 14: Ejemplo ATC

6. Pregunta6: CornersProblem en cornersHeuristic

6.1. Descripción

En este problema, el objetivo es que Pac-Man visite los cuatro puntos de esquina (corners) en un laberinto antes de alcanzar el punto objetivo final. Estos puntos de esquina están ubicados en las esquinas del laberinto y deben ser visitados en cualquier orden antes de llegar al objetivo final.

La función çornersHeuristic.[es] una heurística específica diseñada para guiar a Pac-Man hacia las esquinas del laberinto de manera eficiente teniendo en cuenta el entorno del laberinto,, es decir, un heurístico

Admisible
Solamente necesita:

- Recorrer cada esquina por visitar
- Por cada esquina obtener el camino mas corto entre estas

Es importante recalcar:

6.1.1. Tipo de Algoritmo

- Algoritmo de busqueda **no informado**

6.2. Algoritmo

```
(u)
    MIENTRAS haya esquinas por visitar
        Coste=Distancia entre Pac-Man y esquina
        MIENTRAS haya esquinas por calcular
            esqM,distancia = minDistance(esquinaAct,esquinas)
            esquinaAct=esqM
            Actualizar esquinas
            coste=coste+distancia
        Agregar coste a costes
    heuristico=min(costes)
```

6.2.1. Código V1

Con este codigo conseguimos un 0/3 en el autograder.py

```python
def cornersHeuristic(state, problem):
    """
    A heuristic for the CornersProblem that you defined.

      state:   The current search state
               (a data structure you chose in your search problem)

      problem: The CornersProblem instance for this layout.

    This function should always return a number that is a lower bound on the
    shortest path from the state to a goal of the problem; i.e.  it should be
    admissible (as well as consistent).
    """
    corners = problem.corners  # These are the corner coordinates
    walls = problem.walls  # These are the walls of the maze, as a Grid (game.py)

    "*** YOUR CODE HERE ***"

    def trump(pos, walls, k=1):
        # RECIBE: la posición del pac-man, las paredes y el rango de busqueda
        # DEVUELVE: el número de huecos ocupados en ese cuadrado de búsqueda
        cont = 0
        top, right = walls.height - 2, walls.width - 2

        for i in range(pos[0] -k, pos[0] +k +1):
            if i < 0 or i > right:
                continue

            for j in range(pos[1] -k, pos[1] +k +1):
                if j < 0 or j > top:
                    continue

                # print(f"i: {i}, j: {j}")
                # print(walls[i][j])

                if walls[i][j] is True:
                    cont += 1

        return cont


    # El heurístico a una esquina sería la distancia manhatan de la posición a la esquina más un factor c
    # c -> conteo de los huecos ocupados (walls) alrededor del packman en un área k

    pos = state[0]
    heuristics = []

    for c in corners:

        d = util.manhattanDistance(pos, c)
        t = trump(pos, walls)
        #print(f"Manhatan: {d}, trump: {t}")
        h = d + t

        heuristics.append(h)


    return min(heuristics)
```

Figura 15: Codigo inicial corners heuristic

6.2.2. Código V-Final

```
    "*** YOUR CODE HERE ***"
    def get_lista_sin_elm(elm, lista):
        lista_nueva = []
        for x in lista:
            if x != elm:
                lista_nueva.append(x)
        return lista_nueva

    def get_min(actual, esquinas):
        esq_min = None
        dist = None

        for e in esquinas:
            act_dist = util.manhattanDistance(actual, e)

            if dist is None:
                esq_min = e
                dist = act_dist
            elif act_dist < dist:
                esq_min = e
                dist = act_dist

        return dist, esq_min

    def get_coste_circuito(actual, esquinas_por_cal):
        coste = 0

        while len(esquinas_por_cal) > 0:
            dist, esquina = get_min(actual, esquinas_por_cal)
            esquinas_por_cal = get_lista_sin_elm(esquina, esquinas_por_cal)
            coste += dist
            actual = esquina

        return coste

    costes = []
    pos = state[0]
    esquinas_no_visitadas = state[1]

    for e in esquinas_no_visitadas:
        coste = util.manhattanDistance(pos, e)
        coste += get_coste_circuito(e, get_lista_sin_elm(e, esquinas_no_visitadas))

        costes.append(coste)

    if len(costes) == 0:
        return 0
    return min(costes)
```

```
Provisional grades
==================
Question q1: 3/3
Question q2: 3/3
Question q3: 3/3
Question q4: 3/3
Question q5: 3/3
Question q6: 3/3
Question q7: 0/4
Question q8: 0/3
------------------
Total: 18/25

Your grades are NOT yet registered.  To register
to follow your instructor's guidelines to receive

ag6154lk@msidealan:~/IngInfor/TIA/Labo_1eGela$
```

6.2.3. Problemas y dificultades encarados en el ejercicio

En el comienzo del problema hemos tenido dificultades planteando el problema ya que habían varias dudas sobre que significa que un heurístico sea admisible. Finalmente, gracias a la ayuda de la profesora, hemos entendido el objetivo y comenzado a implementar un heurístico que tenga en cuenta el entorno y contabilizando los huecos libres.Finalmente se ha deshechado ya que resultaba en un heurístico inadmisible.

Finalmente hemos pensado realizar un heurístico que, para cada una de las esquinas por visitar, calcula una distancia. Esta distancia es la suma de las distancias desde la posición actual de Pac-Man a esa esquina, pasando por las otras esquinas en orden de cercanía.

6.2.4. Ejemplo

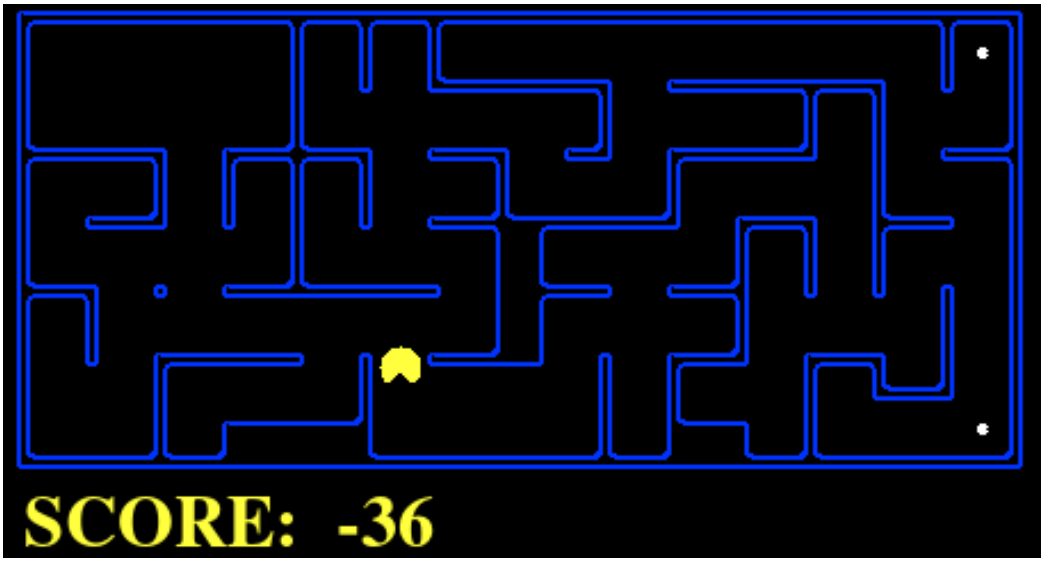

Figura 17: Ejemplo corners heuristic

7. Pregunta7: Eating All The Dots

7.1. Descripción

Este problema consiste en definir un heurístico con el objetivo de comer todos los puntos de comida del mapa sin contar con la presencia de fantasmas o esferas de poder. Este es un problema de búsqueda difícil ya que hay que aportar una solución que recorra todos los puntos de comida evitando

movimientos inecesarios y sin explorar todos los posibles estados. Solamente necesita:

- Recibir la posición del Pac-Man
- Recibir la lista de las comidas

Devuelve:

- El heurístico para ese estado

7.2. Algoritmo

EATING ALL THE DOTS

posicion = PacMan.posicion

listaComidas = PacMan.listaComidas

distancias = []

PARA CADA comida en listaComidas

distancia = Manhatan(posicion, comida)

recorrido = calcurarDistanciaRecorrido(comida, listaComidas)

distancias.push(distancia + recorrido)

return min(distancias)

7.2.1. Código V1

```
def foodHeuristic(state, problem):
    position, foodGrid = state
    foodList = foodGrid.asList()

    dists = []
    for food in foodList:
        dist = util.manhattanDistance(position, food)
        dists.append(dist)

    if len(dists) == 0:
        return 0
    return max(dists)
```

Listing 2: Python example

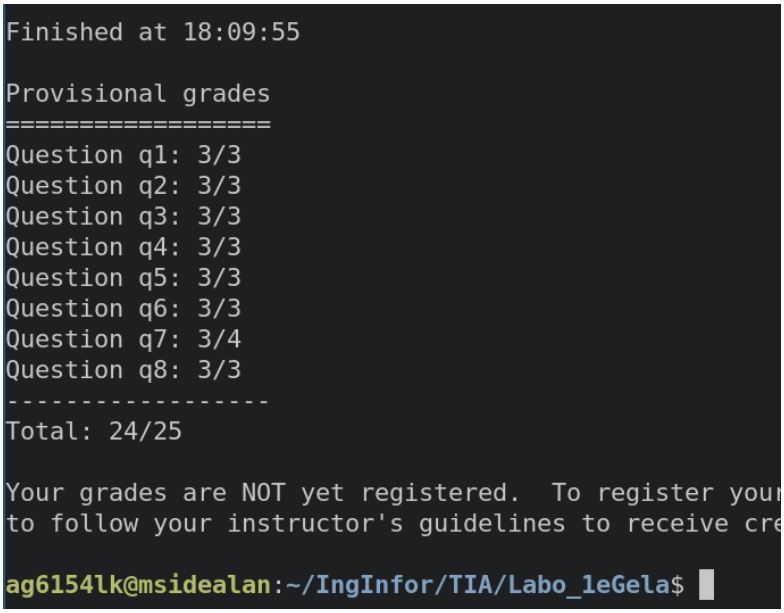

Figura 18: Primera versión Eating All The Dots

7.2.2. Código V-Final

```
def foodHeuristic(state, problem):
    def get_lista_sin_elm(elm, lista):
        lista_nueva = []
        for x in lista:
            if x != elm:
                lista_nueva.append(x)
        return lista_nueva

    def get_min(actual, esquinas):
        esq_min = None
        dist = None

        for e in esquinas:
            act_dist = util.manhattanDistance(actual, e)

            if dist is None:
                esq_min = e
                dist = act_dist
            elif act_dist < dist:
                esq_min = e
                dist = act_dist

        return dist, esq_min

```

```
25      def get_coste_circuito(actual, esquinas_por_cal):
26          coste = 0
27
28          while len(esquinas_por_cal) > 0:
29              dist, esquina = get_min(actual, esquinas_por_cal)
30              esquinas_por_cal = get_lista_sin_elm(esquina,
    esquinas_por_cal)
31              coste += dist
32              actual = esquina
33
34          return coste
35
36      position, foodGrid = state
37      foodList = foodGrid.asList()
38      dists = []
39      esquinas_no_visitadas = state[1]
40
41      for food in foodList:
42          dist = util.manhattanDistance(position, food)
43          dist += get_coste_circuito(food, get_lista_sin_elm(
    food, foodList))
44
45          dists.append(dist)
46
47      if len(dists) == 0:
48          return 0
49      return min(dists)
```

Listing 3: Python example

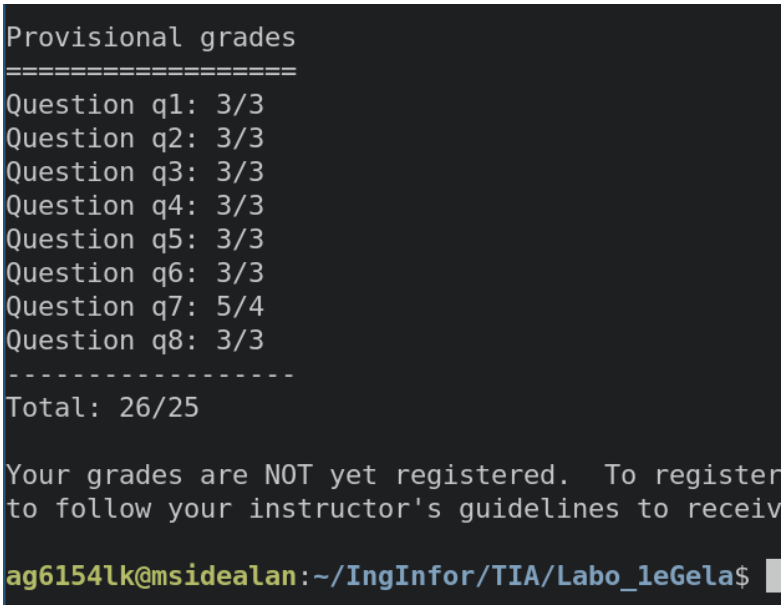

Figura 19: Versión final Eating All The Dots

7.2.3. Problemas y dificultades encarados en el ejercicio

En primer lugar, hemos empleado la distancia a la comida más lejana como heurístico. Con esta solución conseguimos un heurístico admisible que proporcionaba un resultado de 3/4 con la función *autograder.py*.

En un segundo intento por mejorar esta primera solución, hemos recurrido al método *mazeDistance* para el cálculo de las distancias entre el Pac-Man y las comidas, ya que obtenemos unas distancias reales". En esta segunda iteración, siguiendo el mismo principio que en nuestra solución inicial, conseguimos un 5/4 en el autograder. Sin embargo, descartamos esta solución ya que requiere un altísimo coste computacional (por cada comida se calcula la solución al problema *PositionSearchProblem* aplicando BFS) y consideramos que sería hacer trampa porque en un problema de heurística no se debería tener acceso al coste real.

Finalmente, hemos desarrollado un heurístico basándonos en el problema de las esquinas. Así, por cada una de las comidas calculamos la distancia entre el Pac-Man y esa comida, distancia a la cual se le sumará la distancia del recorrido hasta completar todas las comidas yendo desde cada una de ellas a la más cercana, es decir, calculamos la distancia del camino resultado de ir de comida en comida más cercana. Por último, nos quedamos con la distancia mínima de entre todos los caminos posibles. Con esta solución conseguimos un 5/4 en *autograder.py*.

7.2.4. Ejemplo

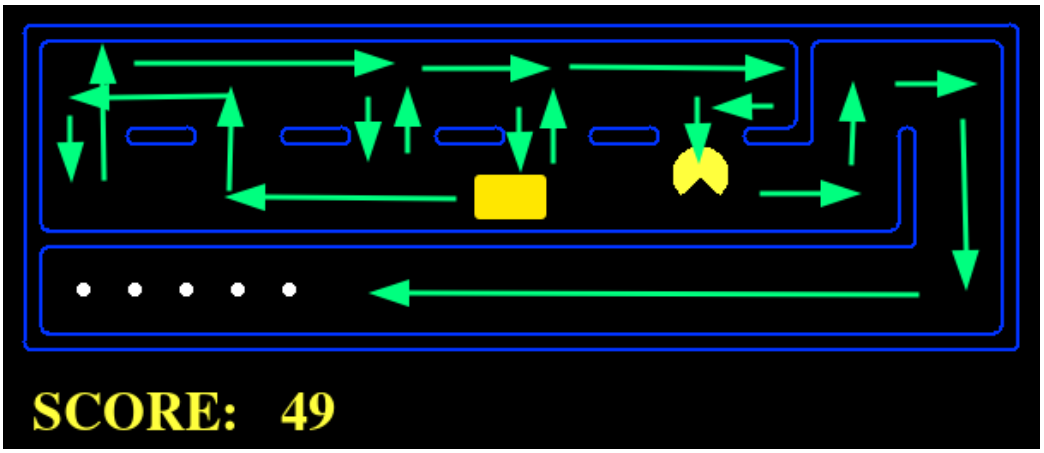

Figura 20: Ejemplo Eating All The Dots

8. Pregunta8: Búsqueda subóptima (SuboptimalSearch)

8.1. Descripción

En un escenario de juego, la búsqueda subóptima basada en ir a la comida más cercana podría implicar que Pac-Man seleccione una dirección que lo lleve a la comida más cercana, sin necesariamente calcular la ruta más corta en términos de movimientos o pasos. Esto podría ser útil en situaciones en las que la prioridad es recolectar comida de manera rápida y efectiva, en lugar de minimizar el número total de movimientos. Solamente necesita:

- Obtener la lista de las comidas
- Recorrer todas las comidas para obtener la más cercana
- Comprobar si el movimiento da con la comida más cercana

8.1.1. Tipo de Algoritmo

- Algoritmo de busqueda **informado**

8.2. Algoritmo

```
(u)
    MIENTRAS haya comidas sin explorar
        distancia = obtener distancia entre comida y estado
        SI distancia ¡distanciaMenor:
            SI nodoAct es meta:
                Actualizar comidaMasCercana y distanciaMenor
    SI estado==comidaMasCercana:
        Goal = True
```

8.2.1. Código V-Final

```
def isGoalState(self, state):
    """
    The state is Pacman's position. Fill this in with a goal test that will
    complete the problem definition.
    """
    x, y = state

    "*** YOUR CODE HERE ***"
    comidaLista=self.food.asList()
    comida_mas_cercana = None
    esGoal=False
    min_distancia=None
    for comida in comidaLista:
        distancia = util.manhattanDistance(state, comida)
        if(min_distancia is None):
            min_distancia = distancia
            comida_mas_cercana = comida
        elif (distancia < min_distancia):
            min_distancia = distancia
            comida_mas_cercana = comida
    if(state==comida_mas_cercana):
        esGoal=True
    return(esGoal)
```

```
def findPathToClosestDot(self, gameState):
    """
    Returns a path (a list of actions) to the closest dot, starting from
    gameState.
    """
    # Here are some useful elements of the startState
    startPosition = gameState.getPacmanPosition()
    food = gameState.getFood()
    walls = gameState.getWalls()
    problem = AnyFoodSearchProblem(gameState)

    acciones= search.bfs(problem)
    return(acciones)
```

Figura 21: Versión final de Suboptimal Search

8.2.2. Problemas y dificultades encarados en el ejercicio

No han habido complicaciones mayores a la hora de implementar este método a excepción de errores mínimos como actualizaciones de variables incorrectas y el formato de un tipo de dato, en especifico al hacer self.food ya que requería ser lista.

8.2.3. Ejemplo

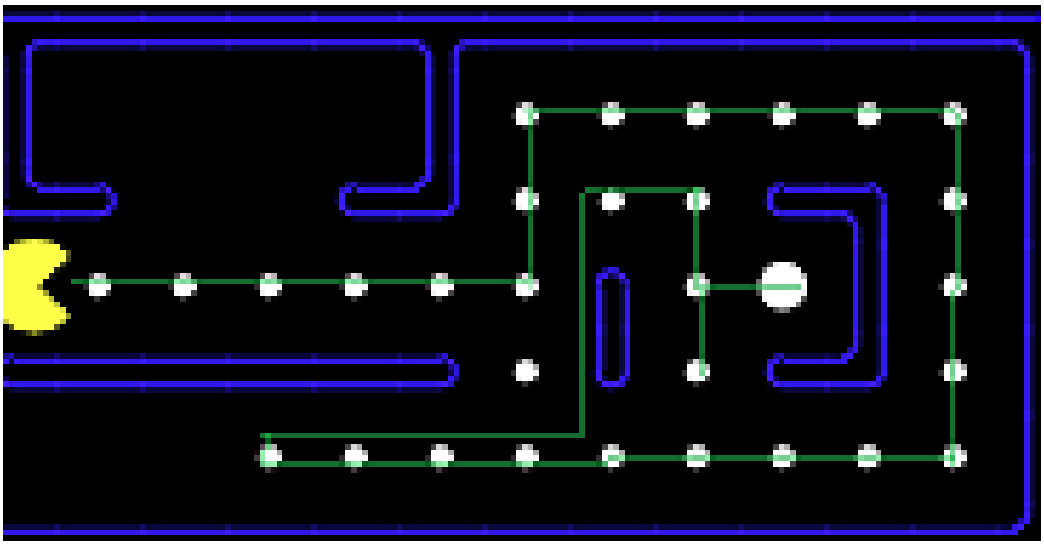

Figura 22: Ejemplo busqueda suboptima