Universidad del País Vasco Euskal Herriko Unibertsitatea
BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

TÉCNICAS DE INTELIGENCIA ARTIFICIAL
4º Curso, Grupo 46
1er cuatrimestre

Practica 1 - Algoritmos de búsqueda

Nagore Gómez y Sergio Martín.

Bilbao, 15 de octubre de 2023

Índice

1. Pregunta 1: Búsqueda En Profundidad

1.1. Descripción

El *Depth First Search* (DFS), o búsqueda en profundidad es un algoritmo de **búsqueda no informada**. Eso quiere decir que es un algoritmo que no emplea ninguna información adicional que no sean los nodos y sus arcos. Si tuviésemos información sobre la meta, por ejemplo su posición, o alguna otra medida que nos indique si vamos bien o mal encaminados en nuestra búsqueda, podríamos emplearla para guiarnos en la búsqueda y entonces estaríamos hablando de una **búsqueda informada**. El DFS es una búsqueda muy primitiva a ese respecto. Simplemente introduce una sistematicidad, una estrategia en la búsqueda. 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 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 el 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 búsqueda **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

```
def depthFirstSearch(problem):

    estadoInicial = problem.getStartState()
    visitados = set()
    pila = [(estadoInicial, [])]

    while len(pila) != 0:
        estadoAct, acciones = pila.pop()

        if problem.isGoalState(estadoAct):
        return acciones

        if estadoAct not in visitados:
            visitados.add(estadoAct)
            sucesores = problem.getSuccessors(estadoAct)

            for sucesor, accion, cost in sucesores:
                if sucesor not in visitados:
                    pila.append(
                        (sucesor, acciones + [accion]))
    return []
```

```
Question q1
===========

*** PASS: test_cases\q1\graph_backtrack.test
***      solution:               ['1:A->C', '0:C->G']
***      expanded_states:        ['A', 'D', 'C']
*** PASS: test_cases\q1\graph_bfs_vs_dfs.test
```

```
***      solution:                 ['2:A->D', '0:D->G']
***      expanded_states:          ['A', 'D']
*** PASS: test_cases\q1\graph_infinite.test
***      solution:                 ['0:A->B', '1:B->C', '1:C->G']
***      expanded_states:          ['A', 'B', 'C']
*** PASS: test_cases\q1\graph_manypaths.test
***      solution:                 ['2:A->B2', '0:B2->C', '0:C->D',
     '2:D->E2', '0:E2->F', '0:F->G']
***      expanded_states:          ['A', 'B2', 'C', 'D', 'E2', 'F']
*** PASS: test_cases\q1\pacman_1.test
***      pacman layout:            mediumMaze
***      solution length: 130
***      nodes expanded:           146

### Question q1: 3/3 ##
```

1.2.2. Código V-Final

```
def depthFirstSearch(problem):
        initialState = problem.getStartState()
            visitedNodes = set()
            unvisitedNodes = util.Stack()

            # Cada elemento es una tupla que tiene un estado y
                las acciones para llegar a el
            unvisitedNodes.push((initialState, []))

            # Recorrer los elementos de la pila
            while not unvisitedNodes.isEmpty():

                # Extraer de la pila el siguiente elemento (
                    estado) + acciones
                actualState, actions = unvisitedNodes.pop()

                # Comprobar si el estado actual es un estado
                    final
                if problem.isGoalState(actualState):
                    # Si es un estado final devolver las
                        acciones para llegar a el
                    return actions

                    # Solo visitar aquellos nodos que no hayan
                        sido visitados antes
                if actualState not in visitedNodes:
```

```
                    # Anadir los nodos que se visitan a la lista
                        de visitados
                    visitedNodes.add(actualState)

                    # Obtener los sucesores del estado actual
                    successors = problem.getSuccessors(
                        actualState)

                    for successor, action, _ in successors: #
                        El coste no se va a usar para nada en
                        este caso

                        # Anadir sucesores con las acciones
                            actualizadas
                        unvisitedNodes.push((successor, actions
                            + [action]))

            # Si no quedan elementos en la cola y no ha habido
                ningun estado final, no hay camino
            return []
```

```
Question q1
===========

*** PASS: test_cases\q1\graph_backtrack.test
***      solution:               ['1:A->C', '0:C->G']
***      expanded_states:        ['A', 'D', 'C']
*** PASS: test_cases\q1\graph_bfs_vs_dfs.test
***      solution:               ['2:A->D', '0:D->G']
***      expanded_states:        ['A', 'D']
*** PASS: test_cases\q1\graph_infinite.test
***      solution:               ['0:A->B', '1:B->C', '1:C->G']
***      expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases\q1\graph_manypaths.test
***      solution:               ['2:A->B2', '0:B2->C', '0:C->D',
     '2:D->E2', '0:E2->F', '0:F->G']
***      expanded_states:        ['A', 'B2', 'C', 'D', 'E2', 'F']
*** PASS: test_cases\q1\pacman_1.test
***      pacman layout:          mediumMaze
***      solution length: 130
***      nodes expanded:         146

### Question q1: 3/3 ###
```

1.2.3. Problemas y dificultades encarados en el ejercicio

Elegir la estructura de datos para almacenar los nodos expandidos pero no explorados puede resultar dudoso, sin embargo, teniendo en cuenta que el siguiente nodo que se debe expandir debe ser el ultimo nodo añadido para ser explorado, se empleará una pila.

Por otro lado, la elección de la estructura para almacenar los nodos ya explorados resulta fácil, se empleará un set. También se podría emplear una lista, pero como el orden de los elementos no es relevante en este caso, se ha optado por el.

Finalmente, la elección de la estructura de datos que contiene cada uno de los elementos de la pila es algo confusa al principio. En ella se ha de guardar en primer lugar el estado en el que se encuentra, seguido de las acciones a tomar para llegar a ese estado desde el estado inicial.

1.2.4. Ejemplo (Figura 1)

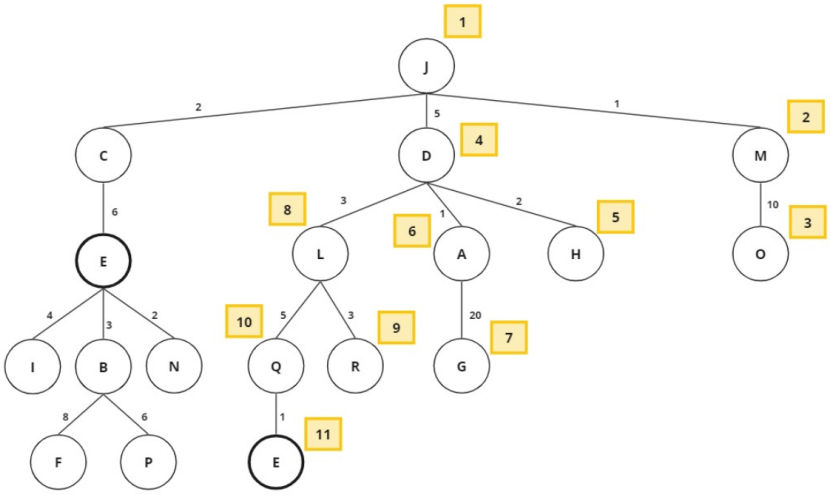

Figura 1: Ejemplo DFS
Expanded nodes: {J, M, O, D, H, A, G, L, R, Q, E}
Path: {J, D, L, Q, E}

2. Pregunta 2: Búsqueda En Anchura

2.1. Descripción

El *Breadth First Search* (BFS), o búsqueda en anchura, es otro algoritmo de **búsqueda no informada**. Al igual que DFS, el BFS tampoco emplea ninguna información adicional que no sean los nodos y sus arcos. Para explorar el grafo, en lugar de seguir profundizando en un camino hasta llegar a un punto final, se mueve gradualmente hacia afuera desde el nodo inicial. Comienza explorando todos los nodos vecinos directos del nodo de inicio antes de pasar a los nodos vecinos de esos nodos, y así sucesivamente. A diferencia del DFS, el BFS puede ser menos eficiente en términos de espacio de memoria cuando se trata de grafos con una gran cantidad de nodos, ya que debe mantener una cola de nodos por explorar que puede crecer significativamente.

Solamente necesita:

- Una función que 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 cola dado que queremos que el siguiente nodo a expandir sea el primer 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.

2.1.1. Tipo de Algoritmo

- Algoritmo de búsqueda **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

```
def breadthFirstSearch(problem):

    estadoInicial = problem.getStartState()
    visitados = set()
    cola = util.Queue()
    cola.push((estadoInicial, []))

    while not cola.isEmpty():
        estadoAct, acciones = cola.pop()

        if problem.isGoalState(estadoAct):
            return acciones

        visitados.add(estadoAct)
        sucesores = problem.getSuccessors(estadoAct)

        for sucesor, accion, cost in sucesores:
            if sucesor not in visitados:
                cola.push((sucesor, acciones + [accion]))

    return []
```

```
Question q2
```

```
==========

*** PASS: test_cases\q2\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases\q2\graph_bfs_vs_dfs.test
***     solution:               ['1:A->G']
***     expanded_states:        ['A', 'B']
*** PASS: test_cases\q2\graph_infinite.test
***     solution:               ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** FAIL: test_cases\q2\graph_manypaths.test
***     graph:
***             B1          E1
***            ^  \        ^  \
***           /    V      /    V
***         *A --> C --> D --> F --> [G]
***           \    ^      \    ^
***            V  /        V  /
***             B2          E2
***
***         A is the start state, G is the goal.  Arrows mark
***         possible state transitions.  This graph has multiple
***         paths to the goal, where nodes with the same state
***         are added to the fringe multiple times before they
***         are expanded.
***     student solution:               ['1:A->C', '0:C->D', '1:
    D->F', '0:F->G']
***     student expanded_states:        ['A', 'B1', 'C', 'B2', '
    C', 'D', 'D', 'E1', 'F', 'E2', 'E1', 'F', 'E2', 'F']
***
***     correct solution:               ['1:A->C', '0:C->D', '1:
    D->F', '0:F->G']
***     correct expanded_states:        ['A', 'B1', 'C', 'B2', '
    D', 'E1', 'F', 'E2']
***     correct rev_solution:           ['1:A->C', '0:C->D', '1:
    D->F', '0:F->G']
***     correct rev_expanded_states:    ['A', 'B2', 'C', 'B1', '
    D', 'E2', 'F', 'E1']
*** FAIL: test_cases\q2\pacman_1.test
*** Too many node expanded; are you expanding nodes twice?
***     student nodes expanded: 275
***
***     correct nodes expanded: 269 (leewayFactor 1.0)
*** Tests failed.

### Question q2: 0/3 ###
```

2.2.2. Código V-Final

```
def breadthFirstSearch(problem):

    initialState = problem.getStartState()
    visitedNodes = set()
    unvisitedNodes = util.Queue()

    # Cada elemento es una tupla que tiene un estado y las
        acciones para llegar a el
    unvisitedNodes.push((initialState, []))

    # Recorrer los elementos de la cola
    while not unvisitedNodes.isEmpty():

        # Extraer de la cola el siguiente elemento (estado) +
            acciones
        actualState, actions = unvisitedNodes.pop()

        # Comprobar si el estado actual es un estado final
        if problem.isGoalState(actualState):

            # Si es un estado final devolver las acciones para
                llegar el
            return actions

        # Solo visitar aquellos nodos que no hayan sido
            visitados antes
        if actualState not in visitedNodes:

            # Anadir los nodos que se visitan a la lista de
                visitados
            visitedNodes.add(actualState)

            # Obtener los sucesores del estado actual
            successors = problem.getSuccessors(actualState)

            for successor, action, _ in successors: # El coste
                no se va a usar para nada en este caso

                # Anadir sucesores con las acciones actualizadas
                unvisitedNodes.push((successor, actions + [
                    action]))

    # Si no quedan elementos en la cola y no ha habido ningun
        estado final, no hay camino
    return []
```

```
Question q2
===========

*** PASS: test_cases\q2\graph_backtrack.test
***      solution:               ['1:A->C', '0:C->G']
***      expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases\q2\graph_bfs_vs_dfs.test
***      solution:               ['1:A->G']
***      expanded_states:        ['A', 'B']
*** PASS: test_cases\q2\graph_infinite.test
***      solution:               ['0:A->B', '1:B->C', '1:C->G']
***      expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases\q2\graph_manypaths.test
***      solution:               ['1:A->C', '0:C->D', '1:D->F', '
    0:F->G']
***      expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1'
    , 'F', 'E2']
*** PASS: test_cases\q2\pacman_1.test
***      pacman layout:          mediumMaze
***      solution length: 68
***      nodes expanded:         269

### Question q2: 3/3 ###
```

2.2.3. Problemas y dificultades encarados en el ejercicio

El cambio de DFS a BFS es prácticamente inmediato, pues lo único que requiere es un cambio en la estructura de datos empleada para guardar los sucesores a explorar.

Con el *Código V1* se consigue una puntuación de 0 en el *autograder* debido a la posición de la sentencia condicional que comprueba si los sucesores han sido visitados o no. Cambiando el orden de las comprobaciones se consigue la puntuación máxima.

2.2.4. Ejemplo (Figura 2)

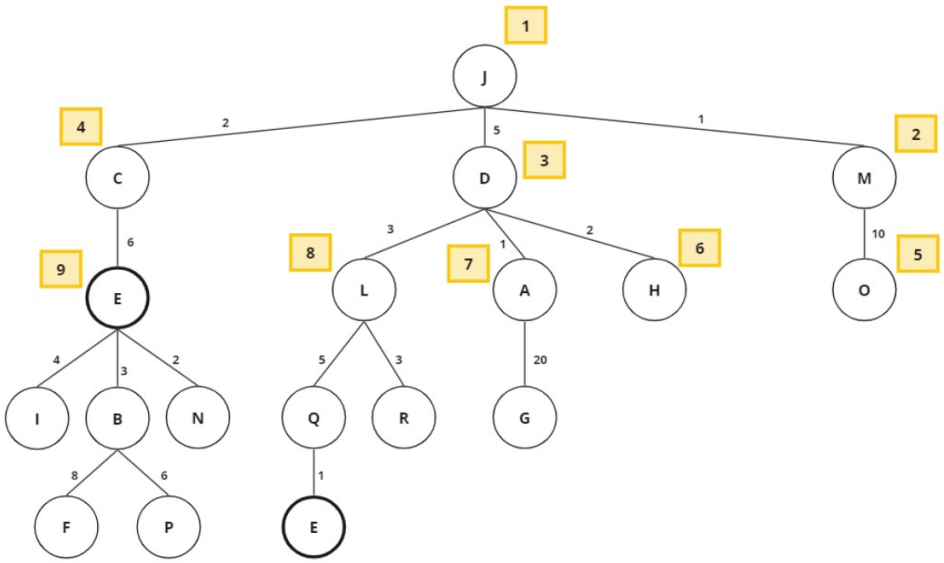

Figura 2: Ejemplo BFS
Expanded nodes: {J, M, D, C, O, H , A, L, E}
Path: {J, C, E}

3. Pregunta 3: Cambiando la función del coste

3.1. Descripción

El *Uniform Cost Search* (UCS), al igual que DFS y BFS, es un algoritmo de **búsqueda no informado** que se utiliza para encontrar el camino más corto en un grafo ponderado. Sin embargo, a diferencia del BFS, UCS tiene en cuenta los costos asociados a los arcos entre los nodos. El algoritmo UCS comienza desde el nodo inicial y se expande gradualmente hacia los nodos vecinos, eligiendo siempre expandir el nodo con el coste más bajo en cada paso. Esto significa que UCS es especialmente útil cuando se necesita encontrar la ruta más corta en un grafo donde los arcos tienen diferentes costes. A medida que el algoritmo avanza, mantiene una cola de prioridad que contiene los nodos por explorar, ordenados según el coste acumulado hasta ese punto. En caso de aplicar un algoritmo UCS sobre un grafo donde el coste de todos los arcos es el mismo, acabará comportándose como un algoritmo BFS.

Solamente necesita:

- Una función que 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 cola de prioridad dado que queremos que el siguiente nodo a expandir sea el nodo con menor coste acumulado.
- Hace falta almacenar junto con cada nodo el camino desde el comienzo hasta el nodo.
- Hace falta almacenar junto con cada nodo el coste acumulado desde el comienzo hasta el nodo.
- Hace falta asignar a cada valor de la cola un coeficiente de prioridad.
- Hace falta una estructura para almacenar los nodos ya explorados para evitar ciclos.

3.1.1. Tipo de Algoritmo

- Algoritmo de búsqueda **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, coste) in sucesores DO

                Introducir nodoSuc en la frontera (valor de
                    prioridad = coste)
```

3.2.1. Código V1

```
def uniformCostSearch(problem):

    estadoInicial = problem.getStartState()
    visitados = set()
    cola = util.PriorityQueue()
    cola.push((estadoInicial, [], []), 0)

    while not cola.isEmpty():

        estadoAct, acciones, costes = cola.pop()

        if problem.isGoalState(estadoAct):
            return acciones

        if estadoAct not in visitados:
            visitados.add(estadoAct)
            sucesores = problem.getSuccessors(estadoAct)

            for sucesor, accion, cost in sucesores:

                cola.push((sucesor, acciones + [accion], costes
                    + [cost]), sum(costes))
```

```
    return []
```

```
Question q3
===========

*** FAIL: test_cases\q3\graph_backtrack.test
***     graph:
***          B
***          ^
***          |
***         *A --> C --> G
***          |
***          V
***          D
***
***         A is the start state, G is the goal.  Arrows mark
***         possible state transitions.  This tests whether
***         you extract the sequence of actions correctly even
***         if your search backtracks.  If you fail this, your
***         nodes are not correctly tracking the sequences of
***         actions required to reach them.
***     student solution:                   []
***     student expanded_states:            ['A']
***
***     correct solution:                   ['1:A->C', '0:C->G']
***     correct expanded_states:            ['A', 'B', 'C', 'D']
***     correct rev_solution:               ['1:A->C', '0:C->G']
***     correct rev_expanded_states:        ['A', 'B', 'C', 'D']
*** FAIL: test_cases\q3\graph_bfs_vs_dfs.test
***     graph:
***         /-- B
***         |   ^
***         |   |
***         |  *A -->[G]
***         |   |     ^
***         |   V     |
***         \-->D -----/
***
***         A is the start state, G is the goal.  Arrows
***         mark possible transitions
***     student solution:                   []
***     student expanded_states:            ['A']
***
***     correct solution:                   ['1:A->G']
***     correct expanded_states:            ['A', 'B']
***     correct rev_solution:               ['1:A->G']
***     correct rev_expanded_states:        ['A', 'B']
```

```
*** FAIL: test_cases\q3\graph_infinite.test
***     graph:
***          B <--> C
***          ^      /|
***          |     / |
***          V    /  V
***         *A<--/   [G]
***
***         A is the start state, G is the goal.  Arrows mark
***         possible state transitions.
***     student solution:                   []
***     student expanded_states:            ['A']
***
***     correct solution:                   ['0:A->B', '1:B->C', '1:
    C->G']
***     correct expanded_states:            ['A', 'B', 'C']
***     correct rev_solution:               ['0:A->B', '1:B->C', '1:
    C->G']
***     correct rev_expanded_states:        ['A', 'B', 'C']
*** FAIL: test_cases\q3\graph_manypaths.test
***     graph:
***              B1          E1
***             ^  \        ^  \
***            /    V      /    V
***         *A --> C --> D --> F --> [G]
***            \    ^      \    ^
***             V  /        V  /
***              B2          E2
***
***         A is the start state, G is the goal.  Arrows mark
***         possible state transitions.  This graph has multiple
***         paths to the goal, where nodes with the same state
***         are added to the fringe multiple times before they
***         are expanded.
***     student solution:                   []
***     student expanded_states:            ['A']
***
***     correct solution:                   ['1:A->C', '0:C->D', '1:
    D->F', '0:F->G']
***     correct expanded_states:            ['A', 'B1', 'C', 'B2', '
    D', 'E1', 'F', 'E2']
***     correct rev_solution:               ['1:A->C', '0:C->D', '1:
    D->F', '0:F->G']
***     correct rev_expanded_states:        ['A', 'B1', 'C', 'B2', '
    D', 'E1', 'F', 'E2']
*** FAIL: test_cases\q3\ucs_0_graph.test
***     graph:
```

```
***                    C
***                    ^
***                    | 2
***               2    V    4
***          *A <----> B ----> [H]
***                    |1
***               1.5  V  2.5
***            G <---- D ----> E
***                    |
***                  2 |
***                    V
***                   [F]
***
***          A is the start state, F and H is the goal.  Arrows
    mark possible state
***          transitions.  The number next to the arrow is the
    cost of that transition.
***      student solution:                   []
***      student expanded_states:            ['A']
***
***      correct solution:                   ['Right', 'Down', 'Down'
    ]
***      correct expanded_states:            ['A', 'B', 'D', 'C', 'G'
    ]
***      correct rev_solution:               ['Right', 'Down', 'Down'
    ]
***      correct rev_expanded_states:        ['A', 'B', 'D', 'C', 'G'
    ]
*** FAIL: test_cases\q3\ucs_1_problemC.test
*** Solution not correct.
***      student solution length: 0
***      student solution:


***
***      correct solution length: 68
***      correct (reversed) solution length: 68
***      correct solution:
West West West West West West West West West South South East
    East
South South South West West West North West West West West South
     South
South East East East East East East East South South South South
     South
South South West West West West West West West West West West
    West
West West West West West West South West West West West West
    West West
```

```
West West
***      correct (reversed) solution:
West West West West West West West West West South South East
    East
South South South West West West North West West West West South
     South
South East East East East East East East South South South South
     South
South South West West West West West West West West West West
    West
West West West West West West South West West West West West
    West West
West West
*** FAIL: test_cases\q3\ucs_2_problemE.test
*** Solution not correct.
***      student solution length: 0
***      student solution:

***
***      correct solution length: 74
***      correct (reversed) solution length: 74
***      correct solution:
South South West West West West South South East East East East
    South
South West West West West South South East East East East South
    South
West West West West South South East East East East South South
    South
West West West West West West West North West West West West
    West West
West West West West West West West West West West West South
    West West
West West West West West West West
***      correct (reversed) solution:
South South West West West West South South East East East East
    South
South West West West West South South East East East East South
    South
West West West West South South East East East East South South
    South
West West West West West West West North West West West West
    West West
West West West West West West West West West West West South
    West West
West West West West West West West
*** FAIL: test_cases\q3\ucs_3_problemW.test
*** Solution not correct.
```

```
***      student solution length: 0
***      student solution:

***
***      correct solution length: 152
***      correct (reversed) solution length: 152
***      correct solution:
West West West West West West West West West West West West West
     West
West West West West West West West West West West West West West
     West
West West West West West South South South South South South South
    South
South South East East East North North North North North North North
    North
East East South South South South South South East East North
    North
North North North North East East South South South South East
    East
North North East East South South East East East South South
    West West
West West West West South South West West West West West South
    West
West West West West South South East East East East East East
    East
North East East East East East North North East East East East
    East
East South South West West West West South South West West West
    West
West South West West West West West West West West West
***      correct (reversed) solution:
West West West West West West West West West West West West West
     West
West West West West West West West West West West West West West
     West
West West West West West South South South South South South South
    South
South South East East East North North North North North North North
    North
East East South South South South South South East East North
    North
North North North North East East South South South South East
    East
North North East East South South East East East South South
    West West
West West West West South South West West West West West South
    West
```

```
West West West West South South East East East East East East
    East
North East East East East East North North East East East East
    East
East South South West West West West South South West West West
    West
West South West West West West West West West West West
*** FAIL: test_cases\q3\ucs_4_testSearch.test
*** Solution not correct.
***     student solution length: 0
***     student solution:

***
***     correct solution length: 7
***     correct (reversed) solution length: 7
***     correct solution:
West East East South South West West
***     correct (reversed) solution:
West East East South South West West
*** FAIL: test_cases\q3\ucs_5_goalAtDequeue.test
***     graph:
***              1       1       1
***          *A ---> B ---> C ---> [G]
***           |                    ^
***           |          10        |
***           \--------------------/
***
***         A is the start state, G is the goal.  Arrows mark
    possible state
***         transitions.  The number next to the arrow is the
    cost of that transition.
***
***         If you fail this test case, you may be incorrectly
    testing if a node is a goal
***         before adding it into the queue, instead of testing
    when you remove the node
***         from the queue.  See the algorithm pseudocode in
    lecture.
***     student solution:                   []
***     student expanded_states:            ['A']
***
***     correct solution:                   ['1:A->B', '0:B->C', '0:
    C->G']
***     correct expanded_states:            ['A', 'B', 'C']
***     correct rev_solution:               ['1:A->B', '0:B->C', '0:
    C->G']
***     correct rev_expanded_states:        ['A', 'B', 'C']
```

```
*** Tests failed.

### Question q3: 0/3
```

3.2.2. Código V-Final

```
def uniformCostSearch(problem):

    initialState = problem.getStartState()
    visitedNodes = set()
    unvisitedNodes = util.PriorityQueue()

    # Cada elemento es una tupla que tiene otra tupla (estado,
        acciones, costes) y un int (valor de prioridad)
    unvisitedNodes.push((initialState, [], 0), 0)

    # Recorrer los elementos de la cola de prioridad
    while not unvisitedNodes.isEmpty():

        # Extraer de la cola de prioridad el siguiente elemento
            (estado + acciones + costes) + valor de prioridad
        actualState, actions, costs = unvisitedNodes.pop()

        # Comprobar si el estado actual es un estado final
        if problem.isGoalState(actualState):

            # Si es un estado final devolver las acciones para
                llegar el
            return actions

        # Solo visitar aquellos nodos que no hayan sido
            visitados antes
        if actualState not in visitedNodes:

            # Anadir los nodos que se visitan a la lista de
                visitados
            visitedNodes.add(actualState)

            # Obtener los sucesores del estado actual
            successors = problem.getSuccessors(actualState)

            for successor, action, cost in successors:

                # Anadir sucesores con las acciones actualizadas
                unvisitedNodes.push((successor, actions + [
                    action], costs + cost), costs + cost)
```

```
    # Si no quedan elementos en la cola de prioridad y no ha
        habido ningun estado final, no hay camino
    return []
```

```
Question q3
===========

*** PASS: test_cases\q3\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases\q3\graph_bfs_vs_dfs.test
***     solution:               ['1:A->G']
***     expanded_states:        ['A', 'B']
*** PASS: test_cases\q3\graph_infinite.test
***     solution:               ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases\q3\graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '
    0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1'
    , 'F', 'E2']
*** PASS: test_cases\q3\ucs_0_graph.test
***     solution:               ['Right', 'Down', 'Down']
***     expanded_states:        ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases\q3\ucs_1_problemC.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         269
*** PASS: test_cases\q3\ucs_2_problemE.test
***     pacman layout:          mediumMaze
***     solution length: 74
***     nodes expanded:         260
*** PASS: test_cases\q3\ucs_3_problemW.test
***     pacman layout:          mediumMaze
***     solution length: 152
***     nodes expanded:         173
*** PASS: test_cases\q3\ucs_4_testSearch.test
***     pacman layout:          testSearch
***     solution length: 7
***     nodes expanded:         14
*** PASS: test_cases\q3\ucs_5_goalAtDequeue.test
***     solution:               ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']

### Question q3: 3/3 ###
```

3.2.3. Problemas y dificultades encarados en el ejercicio

En este problema de búsqueda se debe tener en cuenta los costes de los arcos, lo que en los dos algoritmos anteriores no era así. Por lo tanto, se debe de modificar la estructura de los elementos que entran en la cola de prioridad. De modo que, los comandos a emplear y las estructuras a introducir en la cola han resultado un poco dudosas al principio.

3.2.4. Ejemplo (Figura 3)

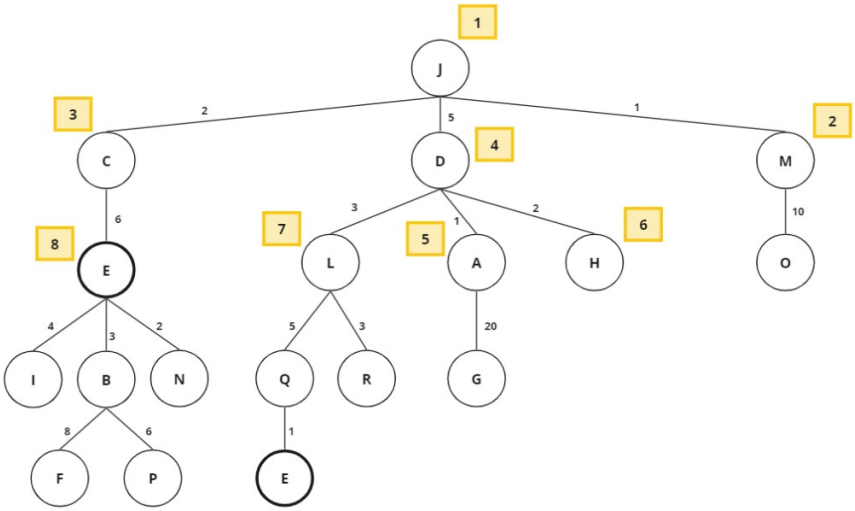

Figura 3: Ejemplo UCS
Expanded nodes: {J, M, C, D, A, H, L, E}
Path: {J, C, E}

4. Pregunta 4: búsqueda *A

4.1. Descripción

El algoritmo de búsqueda A* ("*A star*") es un algoritmo de **búsqueda informado** que se utiliza para encontrar el camino más corto en un grafo ponderado. A diferencia de los algoritmos no informados como el BFS y el DFS, A* emplea información adicional en forma de una **función heurística** para mejorar su eficiencia en la búsqueda de rutas óptimas. El algoritmo de búsqueda A* se distingue por su capacidad para encontrar la ruta más corta en un grafo ponderado utilizando información adicional, en lugar de explorar de manera uniforme como lo hacen el BFS y el DFS. En este caso, la función heurística que se aplica es trivial, por ejemplo, si devuelve siempre el valor 0, el algoritmo terminaría funcionando como un algoritmo UCS. Si además, todos los arcos tuviesen el mismo valor, se convertiría en un algoritmo BFS. A fin de cuentas, el algoritmo A* se comporta de igual forma que el algoritmo UCS, pero el valor de prioridad que emplea se calcula sumando el coste de los arcos y el valor del heurístico.

Solamente necesita:

- Una función que 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*.
- Una función heurística que devuelva un valor heurístico que sea consistente y admisible para cada estado posible.

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 nodo con menor valor de prioridad (coste + heurístico).
- Hace falta almacenar junto con cada nodo el camino desde el comienzo hasta el nodo.
- Hace falta almacenar junto con cada nodo el coste acumulado desde el comienzo hasta el nodo.
- Hace falta asignar a cada valor de la cola un coeficiente de prioridad.
- Hace falta una estructura para almacenar los nodos ya explorados para evitar ciclos.

4.1.1. Tipo de Algoritmo

- Algoritmo de búsqueda **informado**

4.2. Algoritmo

```
A*(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, coste) in sucesores DO

                Calcular el valor del heuristico para el sucesor
                    actual

                Introducir nodoSuc en la frontera (valor de
                   prioridad = heuristico + coste acumulado)
```

4.2.1. Código V1

```
def aStarSearch(problem, heuristic=nullHeuristic):

    estadoInicial = problem.getStartState()
    visitados = set()
    cola = util.PriorityQueue()
    dist_heuristic = heuristic(estadoInicial, problem)
    cola.push((estadoInicial, [], dist_heuristic),
       dist_heuristic)

    while not cola.isEmpty():
        estadoAct, acciones, coste = cola.pop()

        if problem.isGoalState(estadoAct):
            return acciones
```

```
        if estadoAct not in visitados:
            visitados.add(estadoAct)
            sucesores = problem.getSuccessors(estadoAct)

            for sucesor, accion, cost in sucesores:
                dist_heuristic = heuristic(sucesor, problem)
                cola.push((sucesor, acciones + [accion], coste +
                    cost + dist_heuristic), coste + cost +
                   dist_heuristic)

    return []
```

```
Question q4
===========

*** PASS: test_cases\q4\astar_0.test
***     solution:               ['Right', 'Down', 'Down']
***     expanded_states:        ['A', 'B', 'D', 'C', 'G']
*** FAIL: test_cases\q4\astar_1_graph_heuristic.test
***     graph:
***                 2       3       2
***             S ---- A ---- C ---> G
***             | \           /       ^
***           3 |   \ 5     / 1     /
***             |     \     /       /
***             B ---- D ----------/
***                 4           5
***
***         S is the start state, G is the goal.  Arrows mark
    possible state
***         transitions.  The number next to the arrow is the
    cost of that transition.
***
***         The heuristic value of each state is:
***             S 6.0
***             A 2.5
***             B 5.25
***             C 1.125
***             D 1.0625
***             G 0
***     student solution:               ['2', '1', '2']
***     student expanded_states:        ['S', 'A', 'D', 'C', 'B'
    ]
***
***     correct solution:               ['0', '0', '2']
***     correct expanded_states:        ['S', 'A', 'D', 'C']
***     correct rev_solution:           ['0', '0', '2']
```

```
***     correct rev_expanded_states:    ['S', 'A', 'D', 'C']
*** FAIL: test_cases\q4\astar_2_manhattan.test
*** Too many node expanded; are you expanding nodes twice?
***     student nodes expanded: 259
***
***     correct nodes expanded: 221 (leewayFactor 1.1)
*** PASS: test_cases\q4\astar_3_goalAtDequeue.test
***     solution:               ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases\q4\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases\q4\graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '
    0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1'
    , 'F', 'E2']
*** Tests failed.

### Question q4: 0/3 ###
```

4.2.2. Código V-Final

```
def aStarSearch(problem, heuristic=nullHeuristic):

    initialState = problem.getStartState()
    visitedNodes = set()
    unvisitedNodes = util.PriorityQueue()

    # El valor del heuristico sera la distancia desde el estado
        actual hasta el objetivo
    dist_heuristic = heuristic(initialState, problem)

    # Cada elemento es una tupla que tiene otra tupla (estado,
        acciones, costes) y un int (valor de prioridad)
    unvisitedNodes.push((initialState, [], 0), dist_heuristic)

    # Recorrer los elementos de la cola de prioridad
    while not unvisitedNodes.isEmpty():

        # Extraer de la cola de prioridad el siguiente elemento
            (estado + acciones + costes) + valor de prioridad
        actualState, actions, costs = unvisitedNodes.pop()

        # Comprobar si el estado actual es un estado final
        if problem.isGoalState(actualState):
```

```
            # Si es un estado final devolver las acciones para
                llegar el
            return actions

        # Solo visitar aquellos nodos que no hayan sido
            visitados antes
        if actualState not in visitedNodes:

            # Anadir los nodos que se visitan a la lista de
                visitados
            visitedNodes.add(actualState)

            # Obtener los sucesores del estado actual
            successors = problem.getSuccessors(actualState)

            for successor, action, cost in successors:

                # Calcular el valor del heuristico para cada
                    caso concreto
                dist_heuristic = heuristic(successor, problem)

                # Anadir sucesores con las acciones actualizadas
                unvisitedNodes.push((successor, actions + [
                    action], costs + cost), costs + cost +
                    dist_heuristic)

    # Si no quedan elementos en la cola de prioridad y no ha
        habido ningun estado final, no hay camino
    return []
```

```
Question q4
===========

*** PASS: test_cases\q4\astar_0.test
***     solution:               ['Right', 'Down', 'Down']
***     expanded_states:        ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases\q4\astar_1_graph_heuristic.test
***     solution:               ['0', '0', '2']
***     expanded_states:        ['S', 'A', 'D', 'C']
*** PASS: test_cases\q4\astar_2_manhattan.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         221
*** PASS: test_cases\q4\astar_3_goalAtDequeue.test
***     solution:               ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']
```

```
*** PASS: test_cases\q4\graph_backtrack.test
***      solution:                  ['1:A->C', '0:C->G']
***      expanded_states:           ['A', 'B', 'C', 'D']
*** PASS: test_cases\q4\graph_manypaths.test
***      solution:                  ['1:A->C', '0:C->D', '1:D->F', '
    0:F->G']
***      expanded_states:           ['A', 'B1', 'C', 'B2', 'D', 'E1'
    , 'F', 'E2']

### Question q4: 3/3 ###
```

4.2.3. Problemas y dificultades encarados en el ejercicio

El principal problema de Código V1 viene a ser que en la estructura de datos que se emplea en los elementos de la cola de prioridad, se debe guardar el coste real de llegar desde el estado inicial hasta el estado actual, al cual no debería de añadírselo al valor heurístico, pues dejaría de reflejar la realidad. El valor de la función heurística se tendrá en cuenta para el coeficiente de ponderación de la cola de prioridad puesto que eso no representa ningún valor aplicable en la realidad.

4.2.4. Ejemplo (Figura 4)

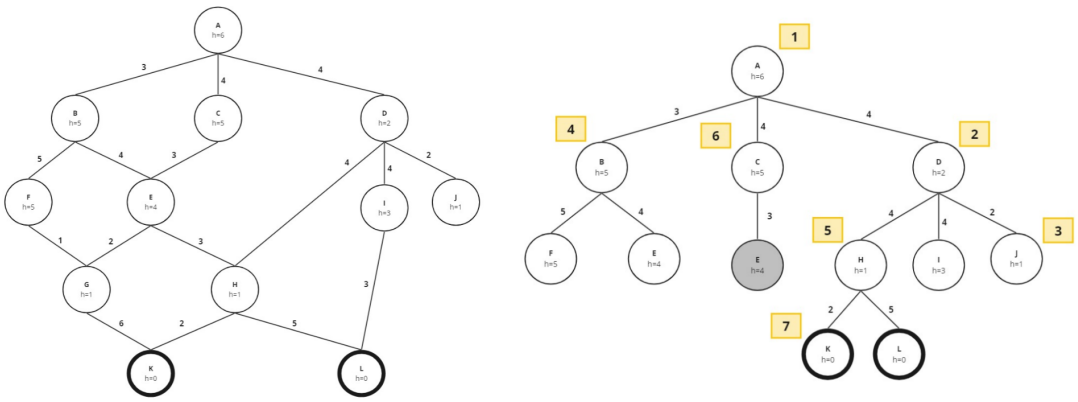

Figura 4: Ejemplo de búsqueda A*
Expanded nodes: {A, D, J, B, H, C, K}
Path: {A, D, H, K}

5. Pregunta 5: Buscando todas las esquinas

5.1. Descripción

El "laberinto de esquinas" es un problema de búsqueda que debe encontrar el camino mas corto a través del laberinto para alcanzar las cuatro esquinas del mismo, haya comida en ellas o no. Para ello, se debe elegir una representación de estado que codifique toda la información necesaria para detectar si se han alcanzado las cuatro esquinas o no. Esa es precisamente la peculiaridad de este ejercicio, se debe de plantear un nuevo estado diferente al anterior porque el problema de búsqueda es distinto.

Solamente necesita:

- Una función que 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 comprueba si el estado "actual" es *meta*.

Es importante recalcar:

- Para este problema, el estado se ha definido como una tupla que contiene, por un lado, las coordenadas de posición, y por otro lado, una lista de booleanos (uno por cada esquina) que indica que esquinas han sido visitadas.

5.1.1. Código V1

```
class CornersProblem(search.SearchProblem):

    def __init__(self, startingGameState, costFn=lambda x: 1):

        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))
        self.costFn = costFn
        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
```

```
        self.visited_corners = set()  # Las coordenadas de
            esquinas visitadas se anadiran aqui

    def getStartState(self):

        state = (self.startingPosition, 0)
        return state

    def isGoalState(self, state):

        if state[1] == 4:
            return True
        else:
            return False

    def getSuccessors(self, state):

        successors = []
        for action in [Directions.NORTH, Directions.SOUTH,
            Directions.EAST, Directions.WEST]:

            xy, corner_count = state
            x = xy[0]
            y = xy[1]
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            if not self.walls[nextx][nexty]:
                if (nextx, nexty) in self.corners and (nextx,
                    nexty) not in self.visited_corners:
                    nextState = ((nextx, nexty), corner_count +
                        1)
                    self.visited_corners.add((nextx, nexty))
                else:
                    nextState = ((nextx, nexty), corner_count)
                cost = self.costFn(nextState)
                successors.append((nextState, action, cost))

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

    def getCostOfActions(self, actions):

        if actions is 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)
```

```
Question q5
===========

*** FAIL: test_cases\q5\corner_tiny_corner.test
*** Corners missed: [(1, 1), (1, 6), (6, 1), (6, 6)]
*** Tests failed.

### Question q5: 0/3 ###
```

5.1.2. Código V-Final

```
class CornersProblem(search.SearchProblem):

    def __init__(self, startingGameState, costFn=lambda x: 1):

        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

        self.costFn = costFn
        self.visitedcorners = (False, False, False, False)

    def getStartState(self):

        return self.startingPosition, self.visitedcorners

    def isGoalState(self, state):

        return all(state[1])

    def getSuccessors(self, state):
```

```
        successors = []
        for action in [Directions.NORTH, Directions.SOUTH,
            Directions.EAST, Directions.WEST]:

            x, y = state[0]
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            if not self.walls[nextx][nexty]:

                nextPos = (nextx, nexty)
                self.visitedcorners = list(state[1])  #
                    Transformar la tupla en lista para modificar

                if nextPos == self.corners[0]:
                    self.visitedcorners[0] = True

                if nextPos == self.corners[1]:
                    self.visitedcorners[1] = True

                if nextPos == self.corners[2]:
                    self.visitedcorners[2] = True

                if nextPos == self.corners[3]:
                    self.visitedcorners[3] = True

                self.visitedcorners = tuple(self.visitedcorners)
                     # Convertir otra vez en tupla

                nextState = nextPos, self.visitedcorners
                cost = self.costFn(nextState)
                successors.append((nextState, action, cost))

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

    def getCostOfActions(self, actions):

        if actions is 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)
```

```
Question q5
===========
```

```
*** PASS: test_cases\q5\corner_tiny_corner.test
***      pacman layout:           tinyCorner
***      solution length:                 28

### Question q5: 3/3 ###
```

5.1.3. Problemas y dificultades encarados en el ejercicio

El principal problema de Código V1 es que el contador que se emplea para anotar el numero de esquinas visitadas no aumenta su valor de manera correcta, siempre oscila entre valores 0 y 1, de tal forma que nunca llega a ser 4 y el problema no se soluciona nunca.

Al contrario, en el Código V-Final, las esquinas visitadas se anotan en el estado en forma de lista de booleanos. De modo que, cuando todos los elementos sean *True* se habrá conseguido el objetivo. También se podría implementar guardando las propias coordenadas de las cuatro esquinas.

6. Pregunta 6: Buscando las esquinas: Heurístico

6.1. Descripción

Un heurístico es una función que se utiliza en problemas de búsqueda que estima el coste de alcanzar un estado objetivo. En situaciones donde la búsqueda exhaustiva o el análisis completo de todas las posibles soluciones no es factible debido a la complejidad del problema, ayudan a simplificar y agilizar la toma de decisiones.

A la hora de contemplar diferentes métricas que pueden ser utilizadas a modo de heurístico se ha de tener en cuenta que el valor devuelto tiene que ser admisible y consistente.

La admisibilidad va estrictamente ligada al coste real, de tal forma que si el valor de el heurístico supera el coste real de realizar una acción concreta, se consideraría un heurístico inadmisible.

Respecto a la consistencia de un heurístico, se considerará consistente si el resultado del heurístico no varia demasiado cuando se aplica sobre contextos similares.

En conclusión, en este apartado se ha de buscar una función heurística que cumpla con los requisitos mencionados anteriormente.

Solamente necesita:

- Una estructura de datos que almacene las coordenadas de las esquinas del escenario de juego.
- Una estructura de datos formada por booleanos que lleve el control de las esquinas visitadas.

Es importante recalcar:

- El heurístico debe de ser admisible, los valores heurísticos deben ser límites inferiores en el coste real de la ruta más corta al objetivo más cercano.
- El heurístico debe de ser consistente, debe suceder que si una acción ha costado c, entonces tomar esa acción solo puede causar una caída en la heurística de a lo sumo c.

6.1.1. Código V1

```
def cornersHeuristic(state, problem):

    pos, visitedCorners = state
    unvisitedCorners = []
    heuristic = 0

    # Obtener las coordenadas de las esquinas que no han sido
        visitadas
    for i in range(4):
        if not visitedCorners[i]:
            unvisitedCorners.append(problem.corners[i])

    # Si no quedan esquinas por visitar el valor del heuristico
        sera 0
    if len(unvisitedCorners) == 0:
        return heuristic

        # Calcular da distancia Manhattan desde la posicion
            actual hasta las esquinas por visitar
    distances = []
    for k in unvisitedCorners:
        distances.append(util.manhattanDistance(pos, k))

    # Encontrar el  ndice  de la esquina mas cercana
    closestCornerIndex = distances.index(min(distances))

    # Encontrar la distancia desde la posicion actual a la
        esquina mas cercana
    closestCornerDistance = distances[closestCornerIndex]

    # Sumar el coste al heuristico
    heuristic = closestCornerDistance

    return heuristic
```

```
Question q6
===========

*** PASS: heuristic value less than true cost at start state
*** PASS: heuristic value less than true cost at start state
*** PASS: heuristic value less than true cost at start state
path: ['North', 'East', 'East', 'East', 'East', 'North', 'North'
    , 'West', 'West', 'West', 'West', 'North', 'North', 'North',
    'North', 'North', 'North', 'North', 'North', 'West', 'West',
    'West', 'West', 'South', 'South', 'East', 'East', 'East', '
```

```
    East', 'South', 'South', 'South', 'South', 'South', 'South',
    'West', 'West', 'South', 'South', 'South', 'West', 'West', '
    East', 'East', 'North', 'North', 'North', 'East', 'East', '
    East', 'East', 'East', 'East', 'East', 'East', 'South', '
    South', 'East', 'East', 'East', 'East', 'East', 'North', '
    North', 'East', 'East', 'North', 'North', 'East', 'East', '
    North', 'North', 'East', 'East', 'East', 'East', 'South', '
    South', 'South', 'South', 'East', 'East', 'North', 'North', '
    East', 'East', 'South', 'South', 'South', 'South', 'South', '
    North', 'North', 'North', 'North', 'North', 'North', 'North',
     'West', 'West', 'North', 'North', 'East', 'East', 'North', '
    North']
path length: 106
*** FAIL: Heuristic resulted in expansion of 1475 nodes

### Question q6: 2/3 ###
```

6.1.2. Código V-Final

```
def cornersHeuristic(state, problem):

    pos, visitedCorners = state
    unvisitedCorners = []
    heuristic = 0

    # Obtener las coordenadas de las esquinas que no han sido
        visitadas
    for i in range(4):
        if not visitedCorners[i]:
            unvisitedCorners.append(problem.corners[i])

    # Si no quedan esquinas por visitar el valor del heuristico
        ser  0
    if len(unvisitedCorners) == 0:
        return heuristic

    # Hacer una iteracion por cada esquina a visitar
    for _ in range(len(unvisitedCorners)):

        # Calcular da distancia Manhattan desde la posicion
            actual hasta las esquinas por visitar
        distances = []
        for k in unvisitedCorners:
            distances.append(util.manhattanDistance(pos, k))

        # Encontrar el  ndice  de la esquina m s cercana
```

```
        closestCornerIndex = distances.index(min(distances))

        # Encontrar la distancia desde la posicion actual a la
            esquina m s cercana
        closestCornerDistance = distances[closestCornerIndex]

        # Sumar el coste al heuristico
        heuristic += closestCornerDistance

        # Actualizar la posicion y remover la esquina visitada
            de unvisitedCorners
        pos = unvisitedCorners[closestCornerIndex]
        del unvisitedCorners[closestCornerIndex]

    return heuristic
```

```
Question q6
===========

*** PASS: heuristic value less than true cost at start state
*** PASS: heuristic value less than true cost at start state
*** PASS: heuristic value less than true cost at start state
path: ['North', 'East', 'East', 'East', 'East', 'North', 'North'
    , 'West', 'West', 'West', 'West', 'North', 'North', 'North',
    'North', 'North', 'North', 'North', 'West', 'West',
    'West', 'West', 'South', 'South', 'East', 'East', 'East', '
    East', 'South', 'South', 'South', 'South', 'South', 'South',
    'West', 'West', 'South', 'South', 'South', 'West', 'West', '
    East', 'East', 'North', 'North', 'North', 'East', 'East', '
    East', 'East', 'East', 'East', 'East', 'East', 'South', '
    South', 'East', 'East', 'East', 'East', 'East', 'North', '
    North', 'East', 'East', 'North', 'North', 'East', 'East', '
    North', 'North', 'East', 'East', 'East', 'East', 'South', '
    South', 'South', 'South', 'East', 'East', 'North', 'North', '
    East', 'East', 'South', 'South', 'South', 'South', 'South', '
    North', 'North', 'North', 'North', 'North', 'North', 'North',
     'West', 'West', 'North', 'North', 'East', 'East', 'North', '
    North']
path length: 106
*** PASS: Heuristic resulted in expansion of 692 nodes

### Question q6: 3/3 ###
```

6.1.3. Problemas y dificultades encarados en el ejercicio

En cuestiones de implementación no se trata de un problema complejo, no obstante, el planteamiento lógico del heurístico si que ha resultado ser mas trabajoso. En un primer momento, como se puede observar en *Código V1* se devolvía la distancia desde la posición "actual" a la esquina mas cercana. Se trata de un calculo de poco coste computacional, sin embargo, es demasiado *cortoplacista* para este problema causando la expansión de demasiados nodos.

El valor del heurístico debe de reflejar una estimación del coste para llegar al objetivo, de modo que, si el valor es la distancia a la esquina mas cercana estaría solo contemplando un cuarto del camino (una de las cuatro esquinas). Se trataría de un heurístico demasiado optimista que expandiría demasiados nodos.

Finalmente, el procedimiento a seguir a sido visitar primero la esquina mas cercana a la posición inicial, y desde este el mas cercano, y así sucesivamente hasta recorrer todas las esquinas. Entonces, el valor que el heurístico toma, es la suma de las distancias calculadas, lo cual es bastante mas realista que lo anterior y por tanto la cantidad de nodos expandidos es menor.

6.1.4. Representación gráfica de la lógica empleada en el heurístico

A continuación se presenta un pequeño diagrama (Figura 5) donde se muestra cual es la lógica empleada para calcular el valor del heurístico. En resumidas cuentas, se calcula cual es la esquina mas cercana al agente y se guarda la distancia a ella, después se actualiza la posición del agente a las coordenadas de la esquina mas cercana encontrada y se busca la siguiente esquina mas próxima desde esa nueva posición.

Una vez se tiene el orden de las esquinas a visitar y las distancias entre si, se suman las distancias para obtener el valor del heurístico. En el diagrama las distancias representadas son euclídeas, mientras que en algoritmo se emplean distancias *manhattan*.

Viendo el diagrama, es evidente que el valor del heurístico nunca superara el coste real, pues al no tener el cuenta los muros, la distancia obtenida en el heurístico sera siempre optimista respecto a la distancia real, y por tanto sera admisible.

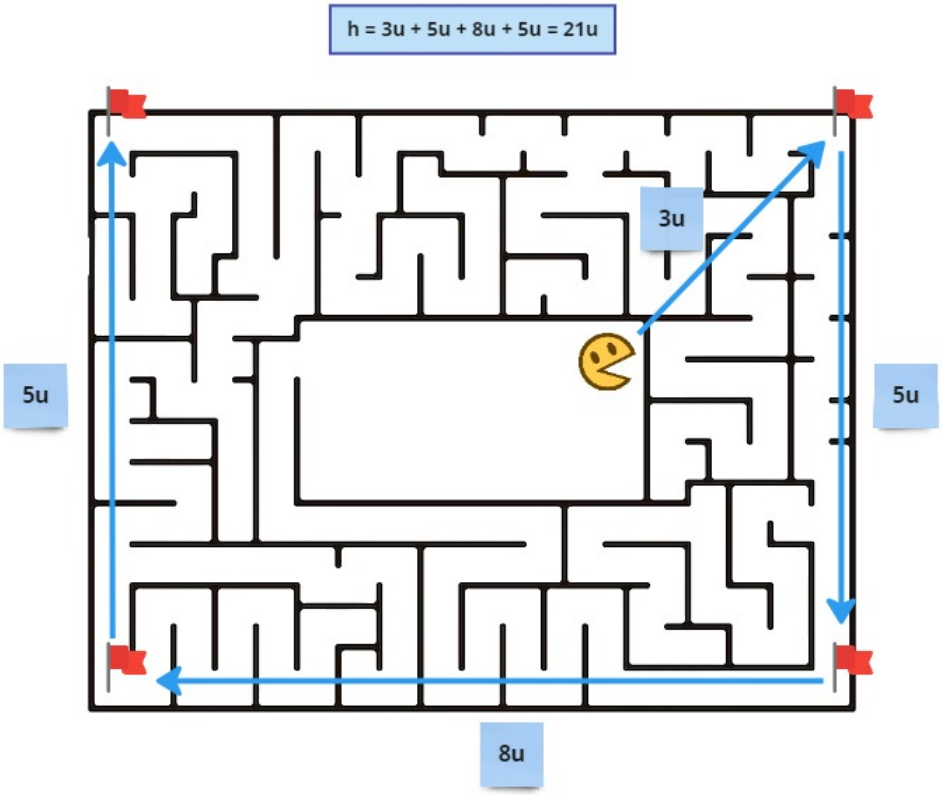

Figura 5: Representación gráfica del heurístico para el problema de las cuatro esquinas

7. Pregunta 7: Comiendo todos los puntos

7.1. Descripción

Comer todos los puntos de comida en el menor número de pasos posible es un problema de búsqueda difícil. Por ello, se debe definir un heurístico que ayude a encontrar el camino mas corto posible para comer todos los puntos en un tiempo razonablemente asequible. Aun usando un algoritmo de búsqueda como $A*$ el problema es demasiado complejo computacionalmente como para ejecutarlo con un heurístico trivial.

Solamente necesita:

- Una estructura de datos que almacene las coordenadas de las comidas que faltan por comer.

Es importante recalcar:

- El heurístico debe de ser admisible, los valores heurísticos deben ser límites inferiores en el coste real de la ruta más corta al objetivo más cercano.
- El heurístico debe de ser consistente, debe suceder que si una acción ha costado c, entonces tomar esa acción solo puede causar una caída en la heurística de a lo sumo c.

7.1.1. Código V1

```
def foodHeuristic(state, problem):

    position, foodGrid = state
    unvisitedFood = foodGrid.asList()

    if len(unvisitedFood) == 0:
        return 0

    heuristic = 0

    for i in range(len(unvisitedFood)):

        distances = []
        for food in unvisitedFood:
            distances.append(util.manhattanDistance(position,
                food))

        closest_food_index = distances.index(min(distances))
```

```
        closest_food_distance = distances[closest_food_index]

        heuristic += closest_food_distance

        position = unvisitedFood[closest_food_index]

        del unvisitedFood[closest_food_index]

    return heuristic
```

```
Question q7
===========

*** PASS: test_cases\q7\food_heuristic_1.test
*** PASS: test_cases\q7\food_heuristic_10.test
*** PASS: test_cases\q7\food_heuristic_11.test
*** PASS: test_cases\q7\food_heuristic_12.test
*** PASS: test_cases\q7\food_heuristic_13.test
*** PASS: test_cases\q7\food_heuristic_14.test
*** FAIL: test_cases\q7\food_heuristic_15.test
*** Heuristic failed admissibility test
*** Tests failed.

### Question q7: 0/4 ###
```

7.1.2. Código V-Final

```
def foodHeuristic(state, problem):

    position, foodGrid = state
    unvisitedFood = foodGrid.asList()

    if len(unvisitedFood) == 0:
        return 0

    heuristic = 0

    for i in range(len(unvisitedFood)):

        distances = []
        for food in unvisitedFood:
            distances.append(util.manhattanDistance(position,
                food))

        food_index = distances.index(max(distances))
```

```
        food_distance = distances[food_index]

        if food_distance > heuristic:
            heuristic = food_distance

        position = unvisitedFood[food_index]
        del unvisitedFood[food_index]

    return heuristic
```

```
Question q7
===========

*** PASS: test_cases\q7\food_heuristic_1.test
*** PASS: test_cases\q7\food_heuristic_10.test
*** PASS: test_cases\q7\food_heuristic_11.test
*** PASS: test_cases\q7\food_heuristic_12.test
*** PASS: test_cases\q7\food_heuristic_13.test
*** PASS: test_cases\q7\food_heuristic_14.test
*** PASS: test_cases\q7\food_heuristic_15.test
*** PASS: test_cases\q7\food_heuristic_16.test
*** PASS: test_cases\q7\food_heuristic_17.test
*** PASS: test_cases\q7\food_heuristic_2.test
*** PASS: test_cases\q7\food_heuristic_3.test
*** PASS: test_cases\q7\food_heuristic_4.test
*** PASS: test_cases\q7\food_heuristic_5.test
*** PASS: test_cases\q7\food_heuristic_6.test
*** PASS: test_cases\q7\food_heuristic_7.test
*** PASS: test_cases\q7\food_heuristic_8.test
*** PASS: test_cases\q7\food_heuristic_9.test
*** FAIL: test_cases\q7\food_heuristic_grade_tricky.test
***     expanded nodes: 8763
***     thresholds: [15000, 12000, 9000, 7000]

### Question q7: 4/4 ###
```

7.1.3. Problemas y dificultades encarados en el ejercicio

En *Código V1* se prueba a emplear el heurístico planteado en el problema anterior, pero para problema de búsqueda, resulta ser inadmisible.

El heurístico empleado finalmente consiste en lo siguiente: Se debe encontrar la comida mas lejana desde la posición actual y desde esa comida la distancia a la comida mas lejana, y así sucesivamente hasta visitar todas las comidas. El heurístico se ha definido como la distancia máxima entre las distancias calculadas anteriormente, de esta manera, tiende a

ser más optimista en su estimación del costo real para llegar al objetivo.

7.1.4. Representación gráfica de la lógica empleada en el heurístico

Para el problema de *Eat All Dots* la lógica a seguir es un poco diferente al caso anterior, como se puede ver en el diagrama (Figura 6). Esta vez se buscara siempre la comida mas distante a la posición del agente, y como en el caso anterior, se actualizara la posición del *pacman* a la localización de esa comida. Este proceso se repetirá sucesivamente hasta haber alcanzado todas las comidas.

El algoritmo conservará todas las distancias calculadas y devolverá como valor heurístico la máxima entre todas ellas. De modo que, el valor devuelto siempre sera inferior a la distancia que supone alcanzar todos los puntos de comida, pues la distancia máxima corresponde únicamente a la separación entre los dos puntos de comida mas alejados entre si. Nuevamente, se trabaja con distancias *manhattan* y se obtiene un heurístico admisible.

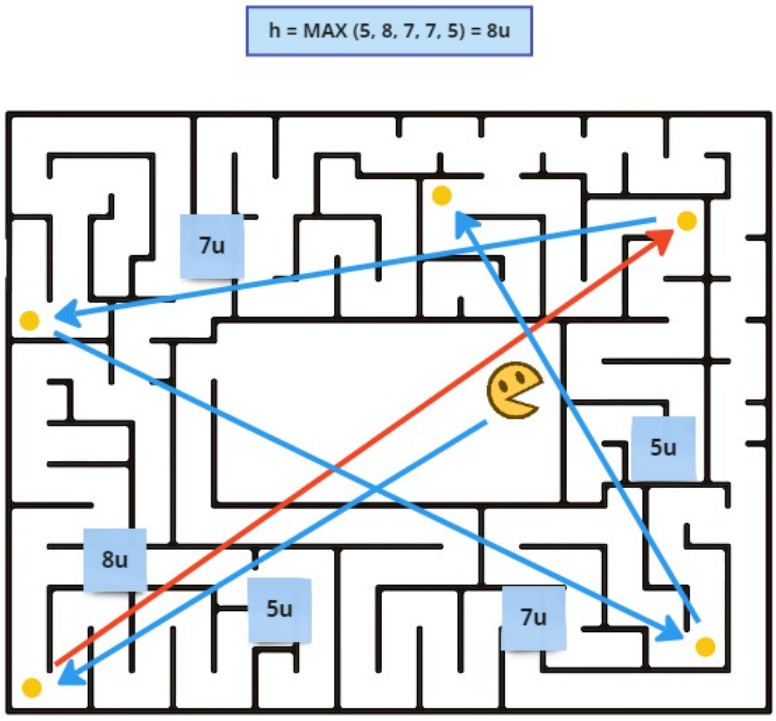

Figura 6: Representación gráfica del heurístico para el problema *Eat All Dots*

8. Pregunta 8: Búsqueda subóptima

8.1. Descripción

El problema de búsqueda subóptima debe encontrar un camino razonablemente bueno, utilizando un agente que coma con voracidad el punto mas cercano. Se trata de encontrar una solución mas simple, aunque menos eficiente, al problema anterior.

Solamente necesita:

- Una función que 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()*).

Es importante recalcar:

- Para encontrar el punto con comida mas cercano el algoritmo mas apropiado seria BFS, para expandir gradualmente las coordenadas alrededor del agente.

8.1.1. Código V1

```
def findPathToClosestDot(self, gameState):

    startPosition = gameState.getPacmanPosition()
    problem = AnyFoodSearchProblem(gameState)

    initialState = startPosition
    visitedNodes = set()
    queue = util.Queue()
    queue.push((initialState, []))

    while not queue.isEmpty():
        actualState, actions = queue.pop()

        if problem.isGoalState(actualState):
            return actions

        if actualState not in visitedNodes:
            visitedNodes.add(actualState)
            successors = problem.getSuccessors(actualState)

            for successor, action, _ in successors:
                queue.push((successor, actions + [action]))
```

```
    return []
```

```
Question q8
===========

[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Warning: this does not look like a regular search maze
*** PASS: test_cases\q8\closest_dot_1.test
***      pacman layout:          Test 1
***      solution length:               1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Warning: this does not look like a regular search maze
*** PASS: test_cases\q8\closest_dot_10.test
***      pacman layout:          Test 10
***      solution length:               1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Warning: this does not look like a regular search maze
*** PASS: test_cases\q8\closest_dot_11.test
***      pacman layout:          Test 11
***      solution length:               2
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Warning: this does not look like a regular search maze
*** PASS: test_cases\q8\closest_dot_12.test
***      pacman layout:          Test 12
***      solution length:               3
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Warning: this does not look like a regular search maze
*** PASS: test_cases\q8\closest_dot_13.test
***      pacman layout:          Test 13
***      solution length:               1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Warning: this does not look like a regular search maze
*** PASS: test_cases\q8\closest_dot_2.test
***      pacman layout:          Test 2
***      solution length:               1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Warning: this does not look like a regular search maze
*** PASS: test_cases\q8\closest_dot_3.test
***      pacman layout:          Test 3
```

```
***      solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Warning: this does not look like a regular search maze
*** PASS: test_cases\q8\closest_dot_4.test
***      pacman layout:          Test 4
***      solution length:                3
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Warning: this does not look like a regular search maze
*** PASS: test_cases\q8\closest_dot_5.test
***      pacman layout:          Test 5
***      solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Warning: this does not look like a regular search maze
*** PASS: test_cases\q8\closest_dot_6.test
***      pacman layout:          Test 6
***      solution length:                2
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Warning: this does not look like a regular search maze
*** PASS: test_cases\q8\closest_dot_7.test
***      pacman layout:          Test 7
***      solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Warning: this does not look like a regular search maze
*** PASS: test_cases\q8\closest_dot_8.test
***      pacman layout:          Test 8
***      solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Warning: this does not look like a regular search maze
*** PASS: test_cases\q8\closest_dot_9.test
***      pacman layout:          Test 9
***      solution length:                1

### Question q8: 3/3 ###
```

8.1.2. Código V-Final

```
def findPathToClosestDot(self, gameState):

    startPosition = gameState.getPacmanPosition()
    problem = AnyFoodSearchProblem(gameState)
```

```
    initialState = startPosition
    visitedNodes = set()
    queue = util.Queue()
    queue.push((initialState, []))

    while not queue.isEmpty():
        actualState, actions = queue.pop()

        if problem.isGoalState(actualState):
            return actions

        if actualState not in visitedNodes:
            visitedNodes.add(actualState)
            successors = problem.getSuccessors(actualState)

            for successor, action, _ in successors:
                queue.push((successor, actions + [action]))

    return []
```

```
Question q8
===========

[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Warning: this does not look like a regular search maze
*** PASS: test_cases\q8\closest_dot_1.test
***      pacman layout:          Test 1
***      solution length:               1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Warning: this does not look like a regular search maze
*** PASS: test_cases\q8\closest_dot_10.test
***      pacman layout:          Test 10
***      solution length:               1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Warning: this does not look like a regular search maze
*** PASS: test_cases\q8\closest_dot_11.test
***      pacman layout:          Test 11
***      solution length:               2
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Warning: this does not look like a regular search maze
*** PASS: test_cases\q8\closest_dot_12.test
***      pacman layout:          Test 12
```

```
***      solution length:               3
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Warning: this does not look like a regular search maze
*** PASS: test_cases\q8\closest_dot_13.test
***      pacman layout:          Test 13
***      solution length:               1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Warning: this does not look like a regular search maze
*** PASS: test_cases\q8\closest_dot_2.test
***      pacman layout:          Test 2
***      solution length:               1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Warning: this does not look like a regular search maze
*** PASS: test_cases\q8\closest_dot_3.test
***      pacman layout:          Test 3
***      solution length:               1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Warning: this does not look like a regular search maze
*** PASS: test_cases\q8\closest_dot_4.test
***      pacman layout:          Test 4
***      solution length:               3
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Warning: this does not look like a regular search maze
*** PASS: test_cases\q8\closest_dot_5.test
***      pacman layout:          Test 5
***      solution length:               1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Warning: this does not look like a regular search maze
*** PASS: test_cases\q8\closest_dot_6.test
***      pacman layout:          Test 6
***      solution length:               2
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Warning: this does not look like a regular search maze
*** PASS: test_cases\q8\closest_dot_7.test
***      pacman layout:          Test 7
***      solution length:               1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Warning: this does not look like a regular search maze
*** PASS: test_cases\q8\closest_dot_8.test
```

```
***      pacman layout:          Test 8
***      solution length:               1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Warning: this does not look like a regular search maze
*** PASS: test_cases\q8\closest_dot_9.test
***      pacman layout:          Test 9
***      solution length:               1

### Question q8: 3/3 ###
```

8.1.3. Problemas y dificultades encarados en el ejercicio

En este apartado no se han dado apenas problemas, simplemente se ha implementado un algoritmo BFS donde los nodos objetivo son aquellos que contienen comida, de modo que el agente se desplaza siempre al punto de comida mas cercano.

8.1.4. Representación gráfica del algoritmo de búsqueda subótima

Mediante BFS se van consultando las posiciones que rodean al agente para encontrar el punto de comida mas cercano. Con el algoritmo BFS se aumenta el rango de búsqueda gradualmente, sin buscar a fondo en cada una de las opciones.

En el diagrama (Figura 7) el rectángulo de color azul representa el rango de análisis del algoritmo de búsqueda.

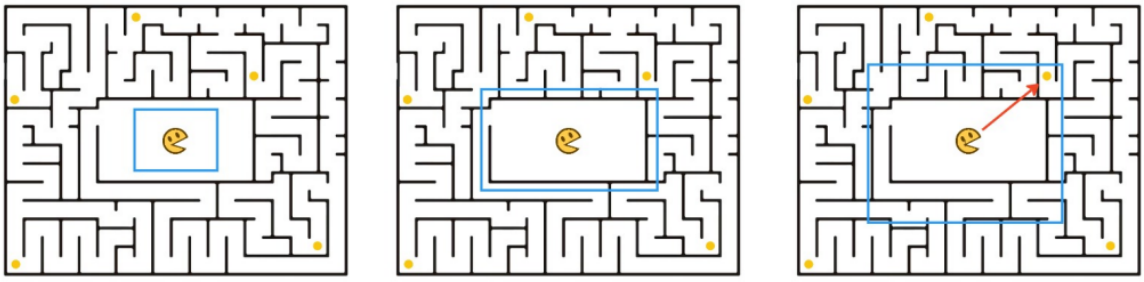

Figura 7: Representación gráfica del algoritmo BFS para *Eat All Dots*