Trabalho 1 - Relatório

Inteligência Artificial

Aluno

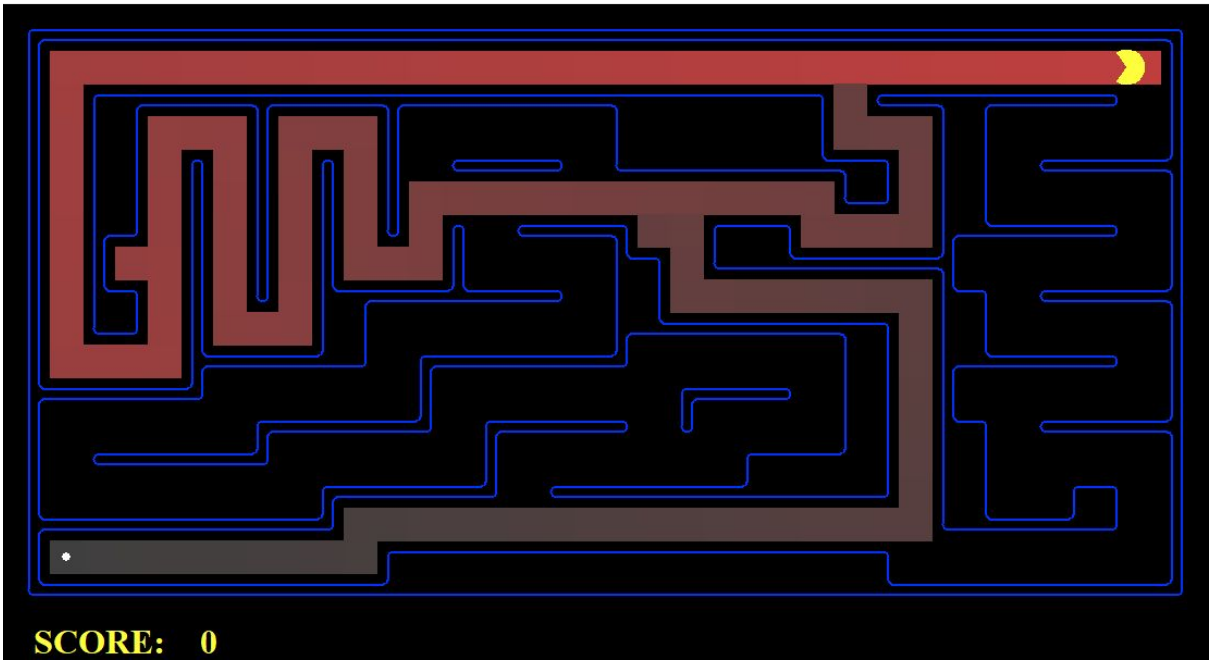
Caio Seda Bittencourt

Período

2021.1

Questão 1

Sim, a ordem de exploração foi de acordo com o esperado. Isso ocorre pois no DFS o algoritmo expande sempre o nó recém adicionado (mais profundo) à Pilha. Como a expansão acontece nas direções Norte, Sul, Leste e Oeste, o caminho se dá indo pela esquerda e não para baixo.



Além disso, no trajeto para o objetivo o pacman não passa por todos os trechos explorados. Isso acontece, pois ao expandir a árvore de busca, um dos caminhos acaba em um estado que já havia sido explorado anteriormente.

Por se tratar de uma busca cega, a prioridade de expansão do algoritmo se dá apenas pela ordem de chegada (último a chegar) dos sucessores do estado atual, preocupando-se apenas em ir o mais longe o mais rápido possível, o que não implica atingir o objetivo. Portanto, esse algoritmo não garante um caminho ótimo.

Questão 2

Diferente do DFS, o BFS encontra a solução ótima. Isso se dá pois, graças ao armazenamento de nós numa fila, o algoritmo garante que todos os nós de um mesmo nível da árvore serão removidos da fronteira. Portanto, nunca haverá nó com menos passos que atinja o objetivo.

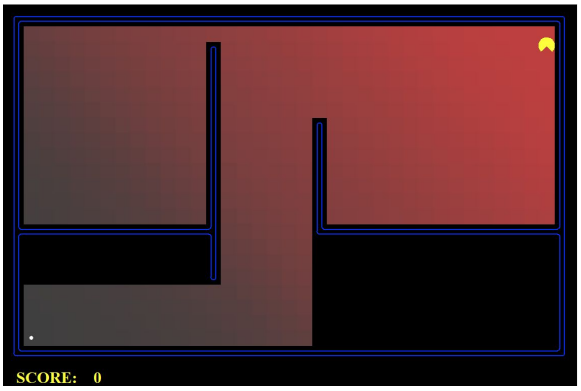
Para o desafio do quebra-cabeças a solução encontrou 8 passos:

```
After 7 moves: up
 -------------
|   | 1 | 2 |
 -------------
| 3 | 4 | 5 |
 -------------
| 6 | 7 | 8 |
 -------------
Press return for the next state...
```

Questão 3

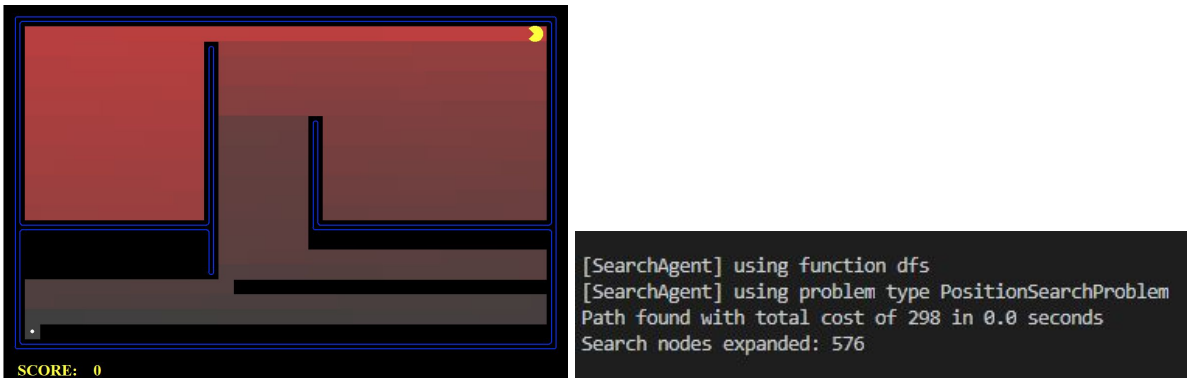
A diferença dos algoritmos no OpenMaze, foi:

Algoritmo	Nós expandidos
A*	535
DFS	576
BFS	682




```
[SearchAgent] using function astar and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 54 in 0.0 seconds
Search nodes expanded: 535
```

*openMaze com A**



openMaze com DFS

openMaze com BFS

Questão 4

O formato utilizado para representar o estado foi `( (x,y), cornerState)`, onde *cornerState* representa uma tupla com 4 valores (um para cada corner) e cada valor podendo variar entre 0 (corner com index correspondente ainda não foi alcançado) e 1 (corner com index correspondente foi alcançado).

```
self.startState = (self.startingPosition, (0, 0, 0, 0))
```

Atributo definido dentro do __init__ de CornersProblem

Para o *getStartState()* ficou definido apenas um *getter* do atributo startState.

```
def getStartState(self):
    """
    Returns the start state (in your state space, not the full Pacman state
    space)
    """
    return self.startState
```

Dentro da função expand, será feito um loop para pegar todas as ações possíveis a partir do nó atual. A partir destas ações, será gerado os estados resultantes e o custo desses estados, em *getNextState* e *getActionCost* respectivamente.

```
def expand(self, state):
    """
    Returns child states, the actions they require, and a cost of 1.

     As noted in search.py:
        For a given state, this should return a list of triples, (child,
        action, stepCost), where 'child' is a child to the current
        state, 'action' is the action required to get there, and 'stepCost'
        is the incremental cost of expanding to that child
    """

    children = []
    for action in self.getActions(state):
        # Add a child state to the child list if the action is legal
        # You should call getActions, getActionCost, and getNextState.
        nextState = self.getNextState(state, action)
        cost = self.getActionCost(state, action, nextState)
        children.append( (nextState, action, cost) )

    self._expanded += 1 # DO NOT CHANGE

    return children
```

A verdadeira mudança ficou dentro da função *getNextState*, onde agora há a chamada de uma outra função, a *getCornerState*. Como o nome diz, agora na função que obtém o novo estado, além de calcular a nova posição a partir da nova ação e o estado antigo, também ocorre a geração da nova tupla contendo os estados de cada corner.

Nesta nova função *getCornerState* ocorre a checagem se a posição atual é igual a um dos corners ou não.

```
def getCornerState(self, state, next_position):
    oldState = list(state[1])
    cornerState = []
    for index, corner in enumerate(self.corners):
        if next_position == corner:
            cornerState.append(1)
        else:
            cornerState.append(oldState[index])

    return tuple(cornerState)

def getNextState(self, state, action):
    assert action in self.getActions(state), (
        "Invalid action passed to getActionCost().")
    x, y = state[0]
    dx, dy = Actions.directionToVector(action)
    nextx, nexty = int(x + dx), int(y + dy)

    next_position = (nextx, nexty)
    corner_state = self.getCornerState(state, next_position)

    return (next_position, corner_state)
```

Por fim, a checagem de objetivo retorna verdadeiro caso todos os valores da tupla contendo os estados do corners sejam 1.

```
def isGoalState(self, state):
    """
    Returns whether this search state is a goal state of the problem.
    """
    # hasCorners = len(state[1]) > 0 # or self.corners
    # return ~hasCorners
    return all(state[1]) # testa se já passou por todos os corners
```

Questão 5

Nesta parte do trabalho, a heurística a seguir pega do *state* todos os corners que ainda não foram atingidos (possuam estado igual a 0). A partir daí é feito uma estimativa de custo com base na distância Manhattan para o *corner* mais próximo (a

menor distância dentre os *corners* disponíveis). Em seguida, de forma recursiva, é feito o mesmo procedimento para estimar a distância a ser percorrida até o corner mais próximo deste atual. Somando assim a distância total estipulada para aquele único estado.

```python
def distanceArgmin(pos, pontos):
    import math
    dist = math.inf
    index = None

    for i, ponto in enumerate(pontos):
        d = util.manhattanDistance(pos, ponto)

        if d < dist:
            index = i
            dist = d

    return dist, index

def cornersHeuristic(state, problem):

    corners = problem.corners # These are the corner coordinates
    walls = problem.walls # These are the walls of the maze, as a Grid (game.py)

    # Pega os corners restantes a partir do estado do corner

    pontos = []
    for index, corner_state in enumerate(state[1]):
        if corner_state == 0:
            ponto = problem.corners[index]
            pontos.append(ponto)

    pos = state[0]
    pathLength = 0

    while pontos:
        dist, index = distanceArgmin(pos, pontos)
        pathLength += dist
        pos = pontos[index]
        pontos.pop(index)

    return pathLength
```

A partir dessa heurística, foi possível alcançar um total de apenas 692 nós expandidos.

```
path length: 106
*** PASS: Heuristic resulted in expansion of 692 nodes

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


Finished at 0:21:45

Provisional grades
==================
Question q3: 4/4
Question q5: 3/3
------------------
Total: 7/7
```

Questão 7

Para concluir essa parte do trabalho foi necessário definir o *isGoalState()* da classe *AnyFoodSearchProblem* como sendo apenas a checagem da matriz contendo as posições da comidas, caso haja comida na posição do pacman ela retornará *True*, do contrário, *False*.

```
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

    return self.food[x][y]
```

Além disso, foi necessário também chamar o algoritmo *A** no método *findPathToClosestDot* da classe *ClosestDotSearchAgent* para obter a lista de ações.

```
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)

    return search.astar(problem)
```

No exemplo a seguir, podemos ver a configuração de um layout que procurar pela comida mais próxima resultará em um caminho subótimo para resolução do problema. Nesta configuração, o algoritmo obteria um caminho com custo 31, sendo que em uma configuração ideal para o mesmo layout, o custo seria 22.