universidade de aveiro deti departamento de electrónica, telecomunicações e informática

Sistemas Inteligentes

2020/2021

Desafio 2, Step 6, 7 e 8 - *CornersHeuristic*, *FoodSearchProblem* e *ClosestDotSearchProblem*

Pedro Matos

84986

pedrolopesmatos@ua.pt

Simão Arrais

85132

simaoarrais@ua.pt

maio de 2021

Índice

Introdução

No âmbito da unidade curricular de Sistemas Inteligentes foi-nos proposto o desenvolvimento de vários algoritmos de pesquisa, como *Depth-first search* (DFS), *Breadth-first search* (BFS), *Uniform-cost search* (UCS) ou *A* Search* e a sua implementação numa versão do jogo do Pac-man, bem como a resolução de problemas que permitissem avaliar se os algoritmos estariam bem desenvolvidos.

As sexta, sétima e oitava etapas deste desafio consistem no desenvolvimento de uma função heurística para o problema *CornersProblem*, e a resolução dos problemas *FoodSearchProblem* e *ClosestDotSearchProblem*.

Corners Heuristic

Tal como tinha sido referenciado no relatório anterior, os resultados obtidos no problema dos cantos utilizando o algoritmo *A** eram piores do que utilizando o algoritmo BFS devido à ausência de uma função heurística não trivial. Desse modo, a primeira etapa deste trabalho foi implementar uma função heurística admissível e consistente. Na Figura 1 podemos observar a implementação da função heurística[1].

```
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)

    xy, visited_corners = state.position, state.visitedCorners
    heuristic_values = [0]

    for corner in corners:
        if corner not in visited_corners:
            # heuristic_values.append(util.manhattanDistance(xy, corner))
            heuristic_values.append(util.euclideanHeuristic(xy, corner))
    return max(heuristic_values)
```

Figura 1 Implementação da função `cornersHeurisitc`

Na realidade, foram estudadas duas funções heurísticas, a “Distância de Manhattan” e a “Distância Euclidiana", seguindo a mesma lógica em ambos os casos: para cada canto não visitado, é calculada a o valor da distância segundo a função utilizada e guardado numa lista,

[1] https://github.com/pedrodlmatos/trabalhos-si/blob/80d5a74557a40bf2b492ced7927e183030712462/desafio2/search/searchAgents.py#L356

retornando, no fim, o valor máximo dessa lista. Para garantir que é retornado sempre um valor, a lista é inicializada com um inteiro de valor 0.

Ambas as funções que calculam a distância são admissíveis porque o valor retornado por elas é menor do que o valor real do custo para atingir o respetivo estado, devido ao facto de não considerarem obstáculos. Para além disso, são consistentes porque em qualquer situação, o valor da distância é sempre superior ou igual a 1 que é valor do custo de cada ação.

Utilizando as funções heurísticas, o agente vai explorar o mapa consoante a localização do canto mais próximo, descrevendo o percurso apresentado na figura abaixo.

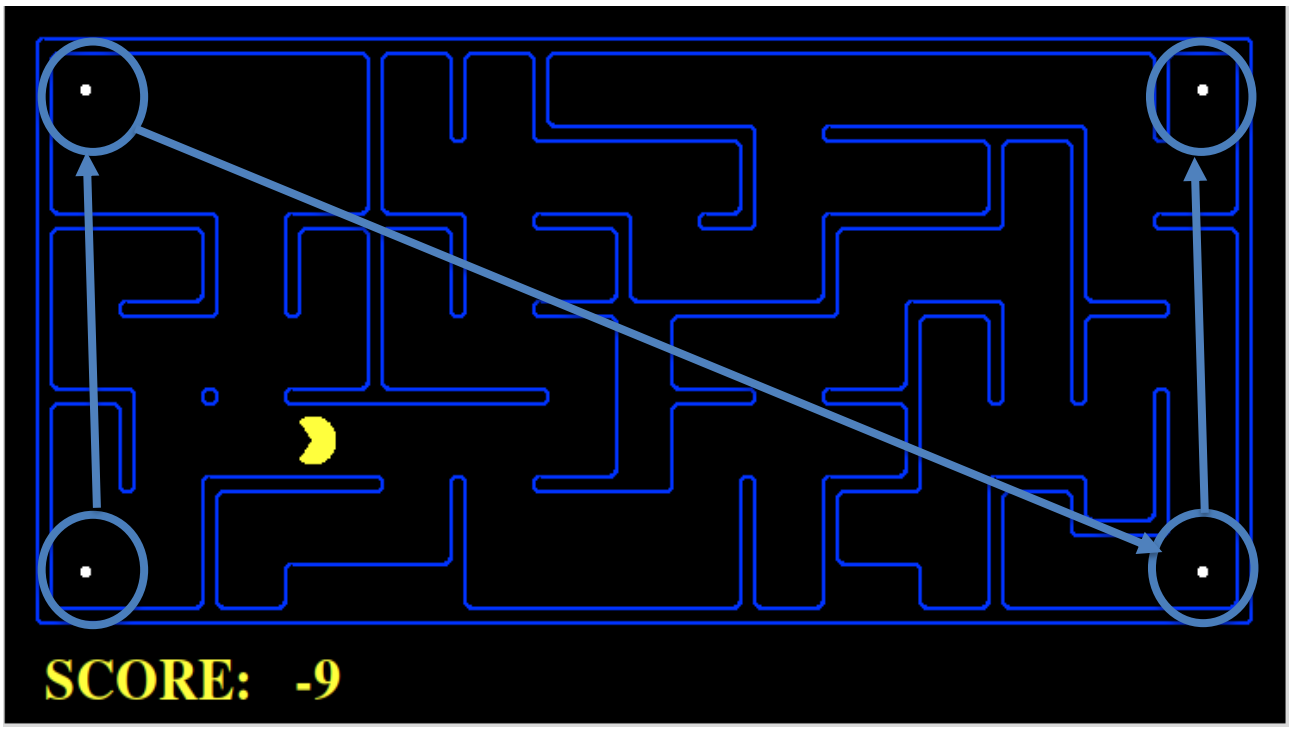

Figura 2 Trajato percorrido pelo agente no problema CornersProblem

O algoritmo DFS, por sua vez, apresenta um percurso bastante diferente devido à sua pesquisa em profundidade, obtendo um custo superior. Na próxima secção iremos apresentar os resultados obtidos para cada algoritmo de pesquisa.

Resultados obtidos

No relatório passado, chegamos à conclusão que uma função heurística trivial não melhorava os resultados obtidos em relação aos do algoritmo BFS. Para melhor percebermos o impacto de uma função heurística não trivial, executamos o agente nos vários mapas e analisamos os resultados. As tabelas 1 e 2 apresentam os resultados obtidos para os mapas *mediumCorners* e *bigCorners*, respetivamente.

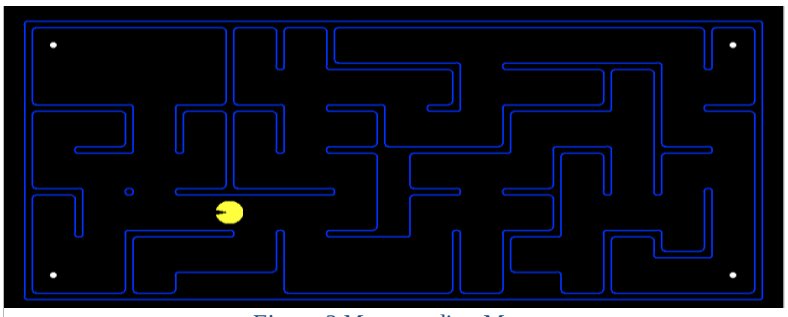

Figura 3 Mapa mediumMaze

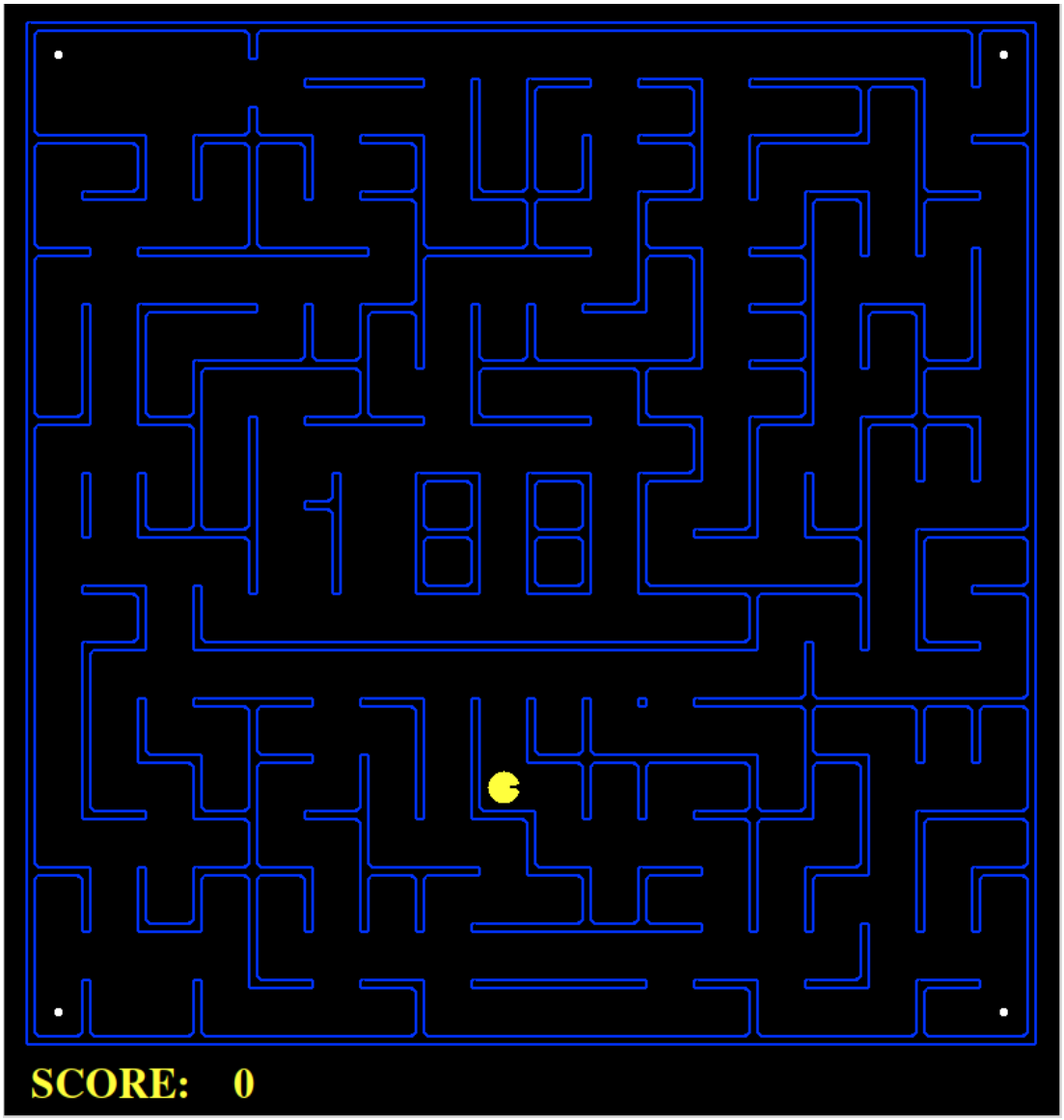

Figura 4 Mapa bigMaze

	Custo	**# Nós expandidos**	**Tempo (s)**
Depth-first Search	221	371	0.1
Breadth-first search	106	1942	1.2
Uniform cost search	106	1966	1.7
A + Distância de Manhattan*	106	1136	0.5
A + Distância Euclidiana*	106	1241	0.6

Tabela 1 Resultados obtidos para o proble CornersProblem *no mapa* mediumMaze

	Custo	# Nós expandidos	Tempo (s)
Depth-first Search	302	504	0.1
Breadth-first search	162	7881	18.2
Uniform cost search	162	7949	19.9
A + Distância de Manhattan*	162	5196	8.7
A + Distância Euclidiana*	162	4380	7.9

Tabela 2 Resultados obtidos para o proble CornersProblem *no mapa* bigMaze

Como seria de esperar, o custo é igual excepto para o algoritmo DFS, e o número de nós expandidos usando o algoritmo *A** já é inferior aos do algoritmo BFS o que demonstra as vantagens que a utilização de uma função heurística podem trazer.

Tendo em conta que o custo corresponde ao tamanho do percurso percorrido, visto que cada ação tem custo igual a 1, pode-se verificar a consistência das funções heurísticas dado que o valor custo é igual ao do algoritmo UCS em ambos os mapas.

A próxima etapa deste desafio é a implementação do problema *FoodSearchProblem* onde o objetivo é passar por todos os pontos disponíveis no mapa no menor percurso possível. A implementação deste problema será apresentada no próximo capítulo.

FoodSearchProblem

O problema *FoodSearchProblem* consiste em descobrir o caminho de menor custo de modo a passar por todos os pontos no mapa. O código de origem fornece uma solução sendo que o objetivo é descobrir um caminho consoante o algoritmo definido. A única função a implementar é a função heurística[2] para utilizar com o algoritmo *A** que está presente na figura seguinte.

[2] https://github.com/pedrodlmatos/trabalhos-si/blob/80d5a74557a40bf2b492ced7927e183030712462/desafio2/search/searchAgents.py#L447

```python
def foodHeuristic(state, problem):
    """Your heuristic for the FoodSearchProblem goes here...."""
    position, foodGrid = state
    heuristic_values = [0]
    for food in foodGrid.asList():
        # heuristic = mazeDistance(position, food, problem.startingGameState)
        heuristic = util.manhattanDistance(position, food)
        # heuristic = util.euclideanHeuristic(position, food)
        heuristic_values.append(heuristic)
    return max(heuristic_values)
```

Figura 5 Implementação da função `foodHeuristic`

Como podemos verificar, utilizamos várias funções para calcular a distância que separa o agente dos vários pontos (`food`). Para cada ponto é calculado a distância e o seu valor é guardado numa lista e, à semelhança da etapa anterior, é retornado o valor máximo presente na lista após iterar por todos os pontos.

Os resultados obtidos serão apresentados na próxima secção.

Resultados obtidos

Neste problema, à semelhança do anterior, foram utilizados mapas apropriados para estudar a implementação da função heurística. No entanto, começamos por avaliar se os algoritmos de pesquisa foram corretamente desenvolvidos explorando o mapa *testSearch* com os algoritmos de pesquisa *A** com uma função heurística trivial e UCS. Os resultados obtidos em ambos os casos estão presentes na tabela abaixo.

Figura 6 Mapa testSearch

	Custo	# Nós expandidos	Tempo (s)
Uniform Cost Search	7	14	0.0
A + heurística trivial*	7	14	0.0

Tabela 3 Resultados obtidos para o problema FoodSearchProblem *no mapa* testSearch

Como seria de esperar, os resultados foram os mesmos o que mostra que o funcionamento do algoritmo *A** com uma heurística trivial é equivalente ao do algoritmo UCS. No entanto, para mapas mais complexos, como o *tinySearch*, o agente começa a demorar mais tempo, como se pode verificar na tabela abaixo.

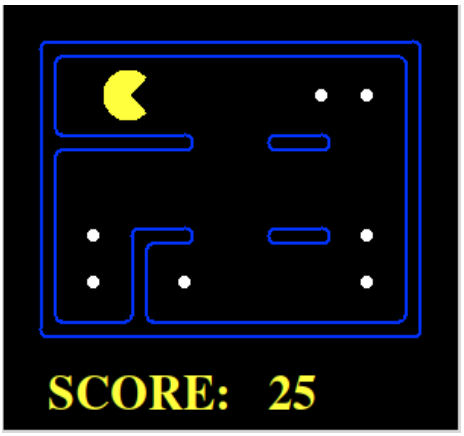

Figura 7 Mapa tinySearch

	Custo	# Nós expandidos	Tempo (s)
Uniform Cost Search	27	5057	4.1
A + heurística trivial*	27	5057	4.0

Tabela 4 Resultados obtidos para o problema FoodSearchProblem *no mapa* tinySearch

Apesar de termos obtido o caminho com o mesmo custo e o mesmo número de nós expandidos comparativamente com a referência que tínhamos, o tempo demorado é superior. No entanto, utilizando as funções heurísticas, o tempo diminui bastante, como se pode verificar na tabela abaixo.

	Custo	# Nós expandidos	Tempo (s)
Depth-first Search	41	59	0.0
Breadth-first search	27	5087	3.9
Uniform cost search	27	5057	4.1
A + heurística trivial*	27	5057	4.0
A + Maze Distance*	27	2372	3.6
A + Distância Euclidiana*	27	2818	1.3
A + Distância de Manhattan*	27	2468	1.1

Tabela 5 Resultados obtidos para o problema FoodSearchProblem *no mapa* tinySearch

A função *mazeDistance*, apesar de explorar um menor número de nós, é a que demora mais tempo, o que se deve ao número superior de operações que realiza. Por fim, decidimos explorar o comportamento do agente no mapa *trickySearch* para obter melhores conclusões relativamente às heurísticas utilizadas.

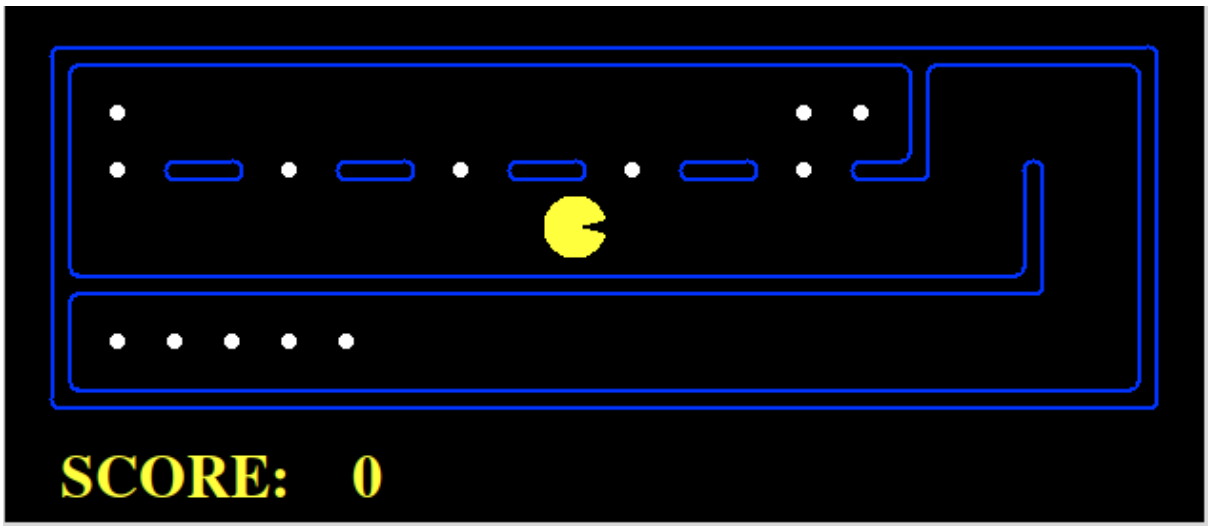

Figura 8 Mapa trickySearch

	Custo	# Nós expandidos	Tempo (s)
Depth-first Search	216	361	0.0
Breadth-first search	60	16518	50.7
Uniform cost search	60	16688	55.8
A + heurística trivial*	60	16688	51.9
A + Maze Distance*	60	4137	21.9
A + Distância Euclidiana*	60	10352	16.5
A + Distância de Manhattan*	60	9551	13.2

Tabela 6 Resultados obtidos para o problema FoodSearchProblem *no mapa* trickySearch

A última etapa deste desafio é implementar o problema *ClosestDotSearchAgent* que vai passando pelo ponto mais perto de si de modo a obter o caminho mais curto que passe por todos os pontos. A implementação do problema e a análise dos resultados obtidos será detalhada no próximo capítulo.

ClosestDotSearchAgent

O problema *ClosestDotSearchAgent* consiste em encontrar o caminho de menor custo de modo a passar por todos os pontos do mapa. O principal objetivo é passar por todos os pontos possíveis o mais rapidamente possível.

Em primeiro lugar, foi necessário implementar a função `findPathToClosestDot`[3] que vai encontar o caminho para o ponto mais próximo no grafo da pesquisa, ou seja, o ponto mais próximo no mapa pode não ser o ponto mais próximo no grafo, como se verifica quando o algoritmo de pesquisa é o DFS. A implementação desta função está presente na Figura 9.

[3] https://github.com/pedrodlmatos/trabalhos-si/blob/80d5a74557a40bf2b492ced7927e183030712462/desafio2/search/searchAgents.py#L447

```python
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.dfs(problem)
    return search.bfs(problem)
    # return search.ucs(problem)
    # return search.astar(problem=problem)
```

Figura 9 Implementação da função `findPathToClosestDot`

A implementação consiste em definir qual o algoritmo de pesquisa dos que foram implementados é utilizado para descobrir o caminho para o ponto mais próximo. A principal desvantagem desta implementação é o facto de apenas ser possível usar uma heurística trivial quando se escolhe o algoritmo *A**, o que não permite verificar as vantagens das funções heurísticas neste problema. Por esse motivo, foi definido o algoritmo BFS para proceder à pesquisa visto que era um dos que encontrava um caminho mais curto.

De seguida, foi necessário implementar a função `isGoalState`[4] que se o estado em que o agente se encontra é um dos pontos do mapa, como se pode ver na Figura 10.

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

Figura 10 Implementação da função `isGoalState`

[4] https://github.com/pedrodlmatos/trabalhos-si/blob/80d5a74557a40bf2b492ced7927e183030712462/desafio2/search/searchAgents.py#L447

Na próxima secção serão discutidos os resultados obtidos para este problema utilizando os mapas *mediumSearch* e *bigSearch*.

Resultados obtidos

Para avaliar o desempenho do agente neste problema, foram utilizados dois mapas, *mediumSearch* e *bigSearch*, em que todos os estados são pontos pelos quais o agente tem que passar.

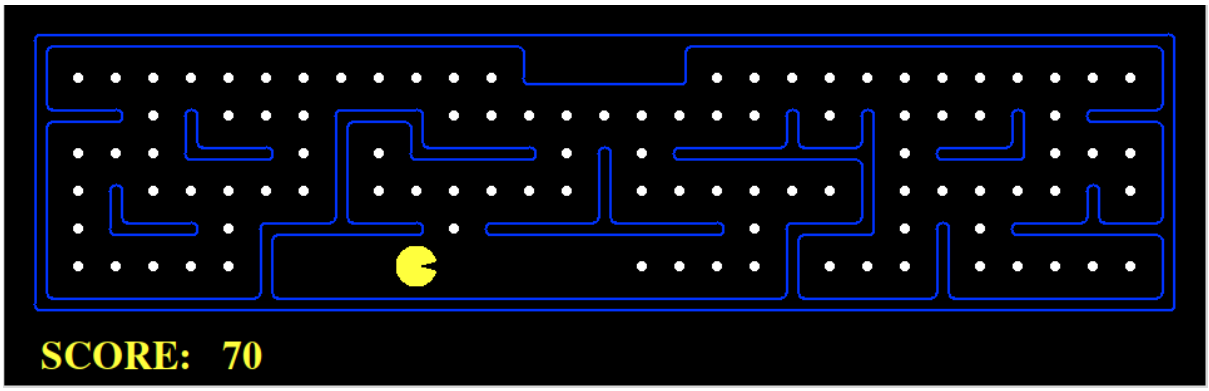

Figura 11 Mapa mediumSearch

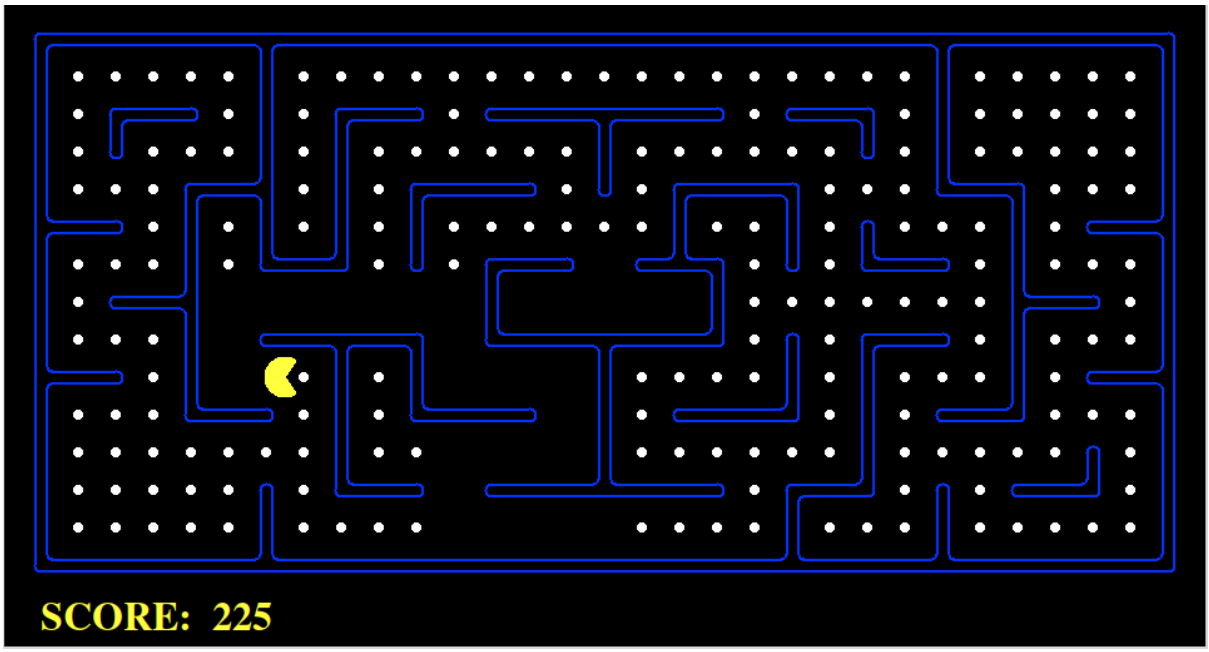

Figura 12 Mapa bigSearch

Os resultados obtidos para os mapas *mediumSearch* e *bigSearch* estão representados nas tabelas 7 e 8, respetivamente.

	Custo	**Score**
Depth-first Search	564	1016
Breadth-first search	190	1410
Uniform cost search	171	1409
A + heurística trivial*	171	1409

Tabela 7 Resultados obtidos para o problema ClosestDotSearchProblem *no mapa* mediumSearch

	Custo	**Score**
Depth-first Search	5324	-2614
Breadth-first search	345	2365
Uniform cost search	350	2360
A + heurística trivial*	350	2360

Tabela 8 Resultados obtidos para o problema ClosestDotSearchProblem *no mapa* bigSearch

Apesar de não nos ser fornecido o tempo que demora a explorar os estados, podemos afirmar que o tempo desde que executamos o comando até o agente se começar a deslocar no mapa é aproximadamente 1 segundo.

No entanto, o tempo que o agente demora a explorar o mapa e a percorrer o caminho definido já é variável consoante o algoritmo escolhido. Esta diferença nota-se particularmente quando o algoritmo DFS é utilizado, o que se pode verificar pelos respetivos valores do custo e do *score*, ou seja, o agente percorre uma distância muito superior "andando" para trás e para a frente, o que não é um caminho ótimo.

Para além disso, podemos verificar que nem sempre seguir o ponto mais próximo significa realizar um percurso mais curto, isto porque pode levar o agente a passar por sítios em que já tinha passado, tornando-se um pouco redundante.

Execução

A classe *main* recebe alguns parâmetros que definem o ambiente onde decorre o jogo. No entanto, para estas etapas do desafio, as definições variam consoante o problema. Todo o código fonte bem como instruções mais detalhadas para executar o projeto podem ser encontradas no repositório[5] no Github.

CornersProblem

Neste problema, existem mapas que são mais apropriados devido à localização dos pontos. Desse modo, o parâmetro que definidos como:

- Mapa, `-l` ou `--layout`:
  - `tinyCorners`
  - `mediumCorners`
  - `bigCorners`
- Agente, `-p` ou `--pacman`:
  - `SearchAgent`
  - `AStarCornersAgent` - forma abreviada de definir o agente como:
    - `-p SearchAgent -a fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic`

Para o caso de se definir o agente como `SearchAgent`, é necessário definir os seus argumentos com:

- Argumento do agente, `-a` ou `-agentArgs`:
  - `fn=dfs,prob=CornersProblem`
  - `fn=bfs,prob=CornersProblem`
  - `fn=ucs,prob=CornersProblem`
  - `fn=astar,prob=CornersProblem,heuristic=cornersHeuristic`

[5] https://github.com/pedrodlmatos/trabalhos-si/tree/main/desafio2

Desse modo, para executar o agente utilizando o algoritmo *A** com a função heurística implementada no mapa *mediumCorners* o comando a ser executado é:

- `python3 pacman.py -l mediumCorners -p AStarCornersAgent` ou
- `python3 pacman.py -l mediumCorners -p SearchAgent -a fn=astar,probCornersProblem,heuristic=cornersHeuristic`

Em ambas situações, o output obtido está presente na figura 13 e ou mapa pelo qual o agente se vai movimentar é dado pela figura 14.

```
Path found with total cost of 106 in 1.1 seconds
Search nodes expanded: 1241
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Figura 13 Exempo de output *obtido no problema* CornersProblem

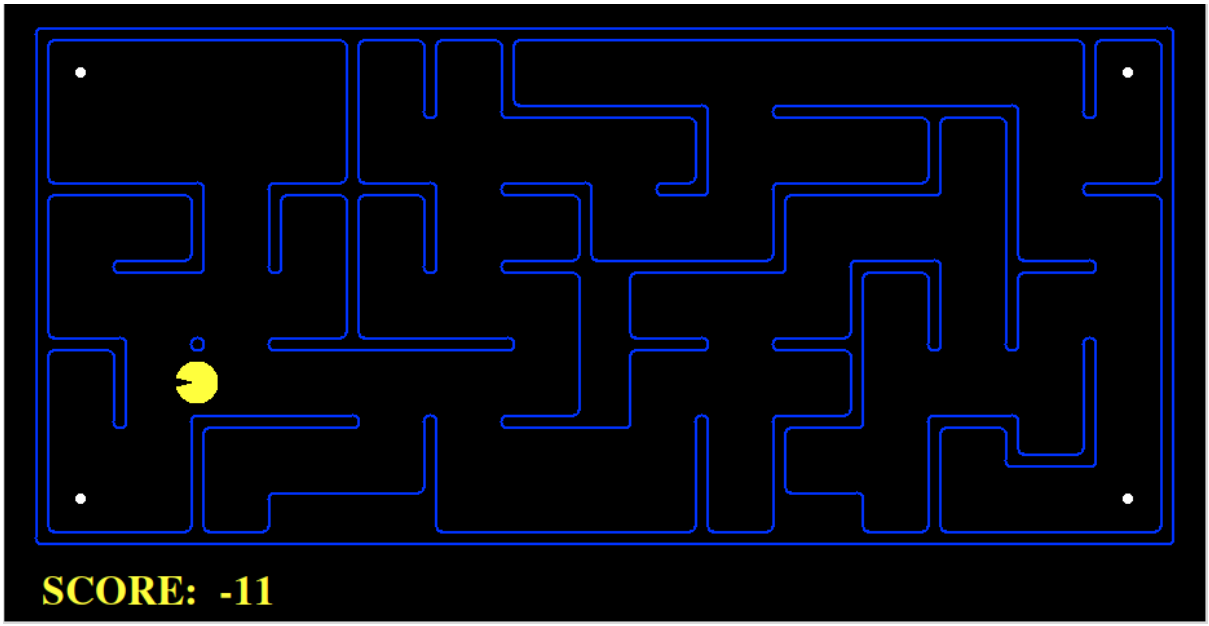

Figura 14 Mapa obtido quando se executa o comando anterior

Por sua vez, para executar o agente no mapa *bigCorners* a pesquisa segundo o algoritmo DFS, o comando a ser executado é:

- `python3 pacman.py -l bigCorners -p SearchAgent -a fn=dfs,prob=CornersProblem`

Nesta situação, tanto o output, como o mapa apresentado serão diferentes, como se pode verificar pelas seguintes imagens.

```
[SearchAgent] using function dfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 302 in 0.2 seconds
Search nodes expanded: 504
Pacman emerges victorious! Score: 238
Average Score: 238.0
Scores:        238.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Figura 15 Exemplo de output *obtido no problema* CornersProblem

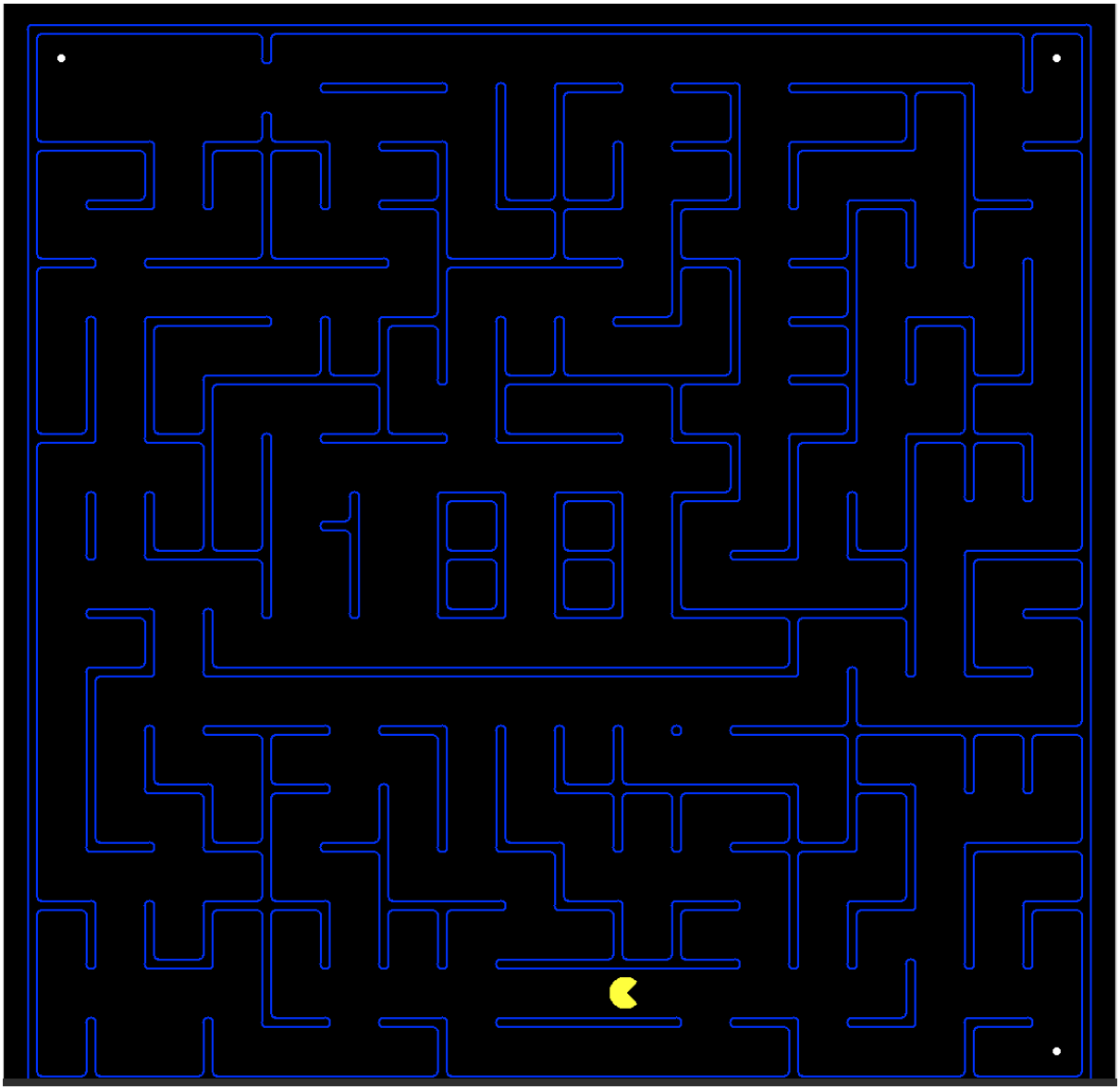

Figura 16 Mapa obtido quando se executa o comando anterior

FoodHeuristic

Para melhor entender o funcionamento das funções heurísticas implementadas, o agente deve movimentar-se por mapas desenvolvidos para esse fim. Desse modo, os parâmetros podem ser definidos como:

- Mapa: `-l` ou `--layout`
  - `testSearch`
  - `tinySearch`
  - `trickySearch`
  - `smallSearch`
- Agente: `-p` ou `--pacman`
  - `SearchAgent`
  - `AStarFoodSearchAgent`
- Argumentos do agente (apenas aplicável quando agente é definido como `SearchAgent`): `-a` ou `--agentArgs`
  - `fn=dfs,prob=FoodSearchProblem`
  - `fn=bfs,prob=FoodSearchProblem`
  - `fn=ucs,prob=FoodSearchProblem`
  - `fn=astar,prob=FoodSearchProblem,heuristic=nullHeuristic`
  - `fn=astar,prob=FoodSearchProblem,heuristic=foodHeuristic`

A última definição dos argumentos do agente pode ser abreviada usando o agente `AStarFoodSearchAgent`. Desse modo, para executar o agente no mapa *tinySearch* utilizando o algoritmo *A** com a função heurística implementada para realizar a pesquisa, o comando a ser executado pode ser:

- `python3 pacman.py -l tinySearch -p AStarFoodSearchAgent` ou
- `python3 pacman.py -l tinySearch -p SearchAgent -a fn=astar,prob=FoodSearchProblem,heuristic=foodHeuristic`

Em ambos os casos, o output obtido e o mapa que vai ser explorado estão nas figuras 17 e 18, respetivamente.

```
[SearchAgent] using function astar and heuristic foodHeuristic
[SearchAgent] using problem type FoodSearchProblem
Path found with total cost of 27 in 2.0 seconds
Search nodes expanded: 2468
Pacman emerges victorious! Score: 573
Average Score: 573.0
Scores:        573.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Figura 17 Exemplo de output *obtido no problema* FoodSearchProblem

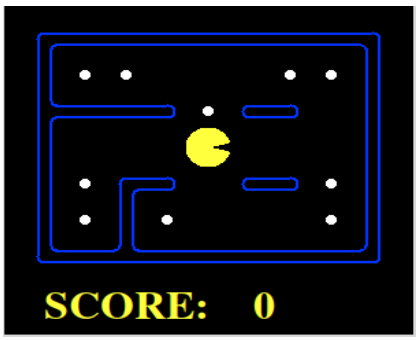

Figura 18 Mapa obtido quando se executa o comando anterior

ClosestDotSearchAgent

Ao contrário dos outros problemas, neste, o agente e os seus argumentos só podem ser definidos de uma forma sendo que para definir o algoritmo de pesquisa é necessário tirar o comentário da respetiva linha e colocar as outras em comentário, à semelhança do que está representado na figura 9.

Desse modo, os parâmetros podem ser definidos como:

- Mapa: `-l` ou `--layout`
  - `mediumSearch`
  - `bigSearch`
- Agente: `-p` ou `--pacman`
  - `ClosestDotSearchAgent`

Assim, para que o agente se desloque pelo mapa *bigSearch* usando o algoritmo UCS é necessário em primeiro lugar remover o comentário do respetivo algoritmo de pesquisa e comentar as restantes e executar o seguinte comando:

- `python3 pacman.py -l bigSearch -p ClosestDotSearchAgent`

O *output* obtido será semelhante ao que está apresentado na figura 19. De notar que, apesar de mostar que o algoritmo de pesquisa definido é o DFS, isso não se verifica na realidade.

```
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with cost 185.
Pacman emerges victorious! Score: 1395
Average Score: 1395.0
Scores:        1395.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Figura 19 Exemplo de output *obtido no problema* ClosestDotSearchAgent

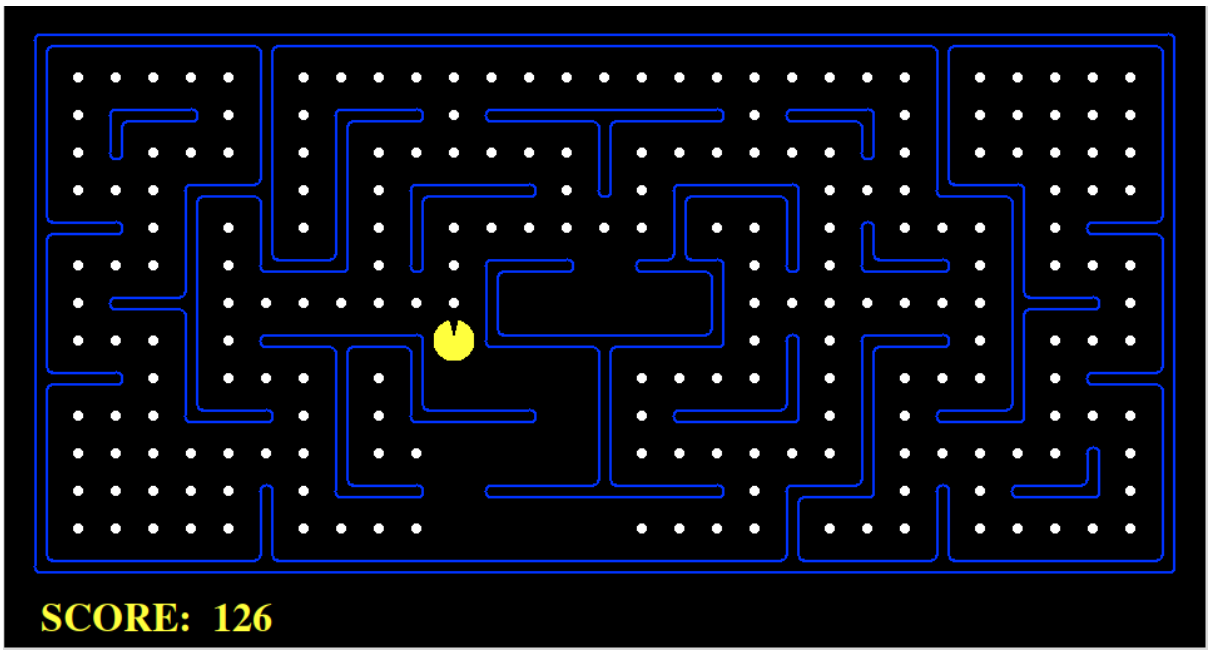

Figura 20 Mapa obtido quando se executa o comando anterior

Conclusão

Estas etapas deste desafio foram importantes para poder avaliar o impacto que as funções heurísticas têm na pesquisa e quais poderão ser realmente vantajosas consoante o problema. Para além disso, foram também importantes para verificar se os algoritmos desenvolvidos nas etapas anteriores estavam corretamente implementados. Na nossa opinião, achamos que os tempos obtidos no problema FoodSearchProblem são demasiado elevados, o que se pode justificar com a estrutura de dados utilizada para guardar os estados visitados e a verificação se um estado já se encontra nessa estrutura. No entanto, por uma questão de lógica, achamos que uma lista é a estrutura mais adequada.