universidade de aveiro deti departamento de electrónica, telecomunicações e informática

Sistemas Inteligentes

2020/2021

Desafio 2, Step 4 e 5 - A* e Problema dos cantos

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** e a sua implementação numa versão do jogo do Pac-man. O principal objetivo deste desafio é analisar as características de cada algoritmo: se é completo ou ótimo, por exemplo, e avaliar as vantagens e desvantagens da sua utilização.

As quarta e quinta etapas deste desafio consistem no desenvolvimento do algoritmo A* e testar a sua implementação no problema dos cantos em que o agente terá que descobrir o melhor caminho de modo a passar por todos os pontos, um em cada canto.

Algoritmo A*

O algoritmo A* é um algoritmo de pesquisa com o principal objetivo de descobrir um caminho que começa num estado inicial e termina num estado final. A principal diferença deste algoritmo em relação aos estudados anteriormente, DFS, BFS e UCS, é que é informado, isto é, utiliza informação adicional para além da informação inerente ao problema, o que o pode levar a obter melhores resultados.

Esta informação adicional é denominada função heurística e é o custo estimado do percurso de menor custo desde um estado inicial até a um estado final. No caso do jogo do Pac-man, o valor do custo de um movimento pode ser calculado como: $f(n) = g(n) + h(n)$ em que $g(n)$ é o valor do custo do problema para atingir o estado n e $h(n)$ é o valor do custo dado pela função heurística.

O algoritmo A* é ótimo e completo mas, em contrapartida, o seu tempo de execução é exponencial e guarda todos os nós em memória, o que em problemas mais complexos pode ser um fator negativo.

Na próxima secção iremos apresentar a implementação do algoritmo A* no jogo do Pac-man.

Implementação

O código fornecido já tinha desenvolvido uma Priority Queue[1] que foi utilizada na implementação do algoritmo para guardar os estados a visitar. Os dados são guardados na estrutura como um tuplo: o primeiro elemento é um tuplo que contém o estado e o caminho até o mesmo, e o segundo elemento é a prioridade (custo) para chegar a esse nó. A estrutura de dados pode ser representada como: ((estado, [direções]), prioridade).

Inicialmente verificamos se o estado inicial é o estado a ser alcançado. Caso não seja, adicionamos esse estado à PriorityQueue com prioridade 0. De seguida, o algoritmo entra num ciclo que vai ser executado até se encontrar a solução ou até já não existirem nós a ser

[1] ficheiro util.py

explorados. Por cada vez que o ciclo se repete, é extraído da PriorityQueue o nó que está à cabeça que vai ser o nó com menor custo.

Se o nó extraído for a estado a ser atingido, então o algoritmo retorna o caminho até o alcançar; caso contrário, se ainda não estiver na lista de estado visitados, é adicionado. De seguida, para cada estado sucessor do estado atual e, se o estado sucessor não estiver na lista de estados visitados, é calculado o custo até o atingir utilizando a função heurística e é adicionado à PriorityQueue com o custo obtido.

Em último lugar, e caso não seja encontrada uma solução, o algoritmo retorna uma lista vazia. Na próxima imagem está presente a implementação do algoritmo descrito acima.

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

    actions = PriorityQueue()
    visited = []                    # list of visited states
    node = problem.getStartState()  # initial node

    # if initial state is the goal state (stop)
    if problem.isGoalState(problem.getStartState()):
        return []

    # push initial state in PriorityQueue
    actions.push((node, []), 0)

    while not actions.isEmpty():
        # get state with lowest cost
        current_state, path_to_state = actions.pop()

        if problem.isGoalState(current_state):
            return path_to_state

        if current_state not in visited:
            visited.append(current_state)
            for successor, direction, cost in problem.getSuccessors(current_state):
                if successor not in visited:
                    path_to_successor = path_to_state + [direction]
                    # calculate cost using heuristic
                    cost_to_successor = problem.getCostOfActions(path_to_successor) + heuristic(successor, problem)
                    actions.push((successor, path_to_successor), cost_to_successor)

    return []
```

*Figura 1 Implementação do algoritmo A**

Resultados Obtidos

Para avaliar os resultados obtidos, executamos o agente nos diversos mapas e utilizamos uma função heurística que não altera o valor do custo alcançar o estado. Observamos os valores obtidos tanto para o custo como para o número de nós expandidos. Os resultados obtidos para o algoritmo desenvolvido estão presentes na tabela abaixo.

Mapa	**Custo**	**# Nós expandidos**	**Tempo (s)**
tinyMaze	8	15	0.0
smallMaze	19	92	0.0
mediumMaze	68	269	0.0
bigMaze	210	620	0.1
openMaze	54	682	0.2

Tabela 1 – Resutados obtidos pelo algoritmo A com a função* nullHeurisitc

No entanto, para avaliar o impacto da função heurística fizemos os mesmos testes mas usando a função heurística *manhattan* que já vinha desenvolvida no código fornecido. Os resultados obtidos estão presentes na tabela abaixo.

Mapa	**Custo**	**# Nós expandidos**	**Tempo (s)**
tinyMaze	8	14	0.0
smallMaze	19	53	0.0
mediumMaze	68	221	0.0
bigMaze	210	549	0.1
openMaze	54	456	0.1

Tabela 2 – Resultados obtidos pelo algoritmo A com a função* manhattanHeurisitc

No mapa *openMaze* observa-se não só a influência que a função heurística tem, visto que o trajeto escolhido é diferente, mas também o comportamento dos diversos algoritmos de pesquisa de forma mais clara.

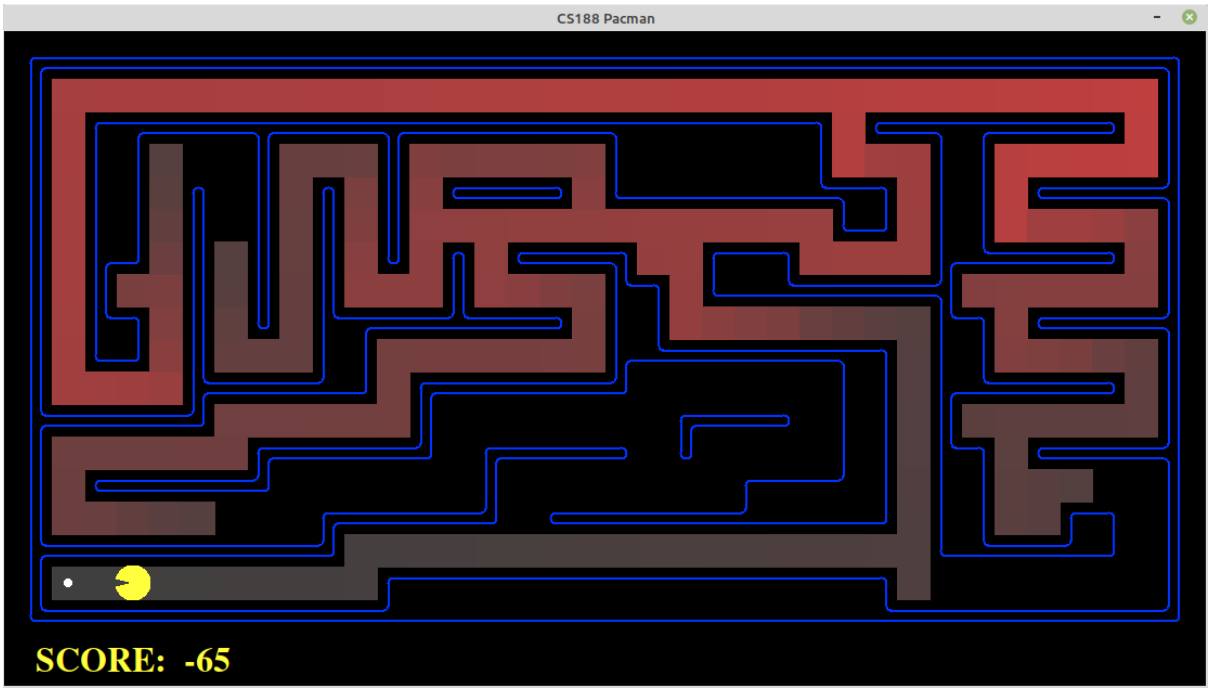

Figura 2 Execução do algoritmo A* *pelo agente no mapa* mediumMaze

Como podemos observar, em termos de custo não existe diferença visto que o agente acaba por percorrer o mesmo caminho mas o número de nós expandidos é menor em todos os mapas o que demonstra que a utilização de funções heurística pode ajudar a obter melhores resultados. Para melhor observar as vantagens deste algoritmo, decidimos comparar os resultados obtidos pelos vários algoritmos nos mapas *mediumMaze* e *bigMaze*.

Algoritmo	**Custo**	**# Nós expandidos**	**Tempo (s)**
Depth-first search	130	146	0.0
Breadth-first search	68	270	0.0
Uniform-cost search	68	269	0.1
*A**	68	221	0.0

Tabela 3 – Resultados obtidos pelos vários algoritmos de pesquisa no mapa mediumMaze

Algoritmo	Custo	# Nós expandidos	Tempo (s)
Depth-first search	210	390	0
Breadth-first search	210	619	0.1
Uniform-cost search	210	620	0.2
A*	210	549	0.1

Tabela 4 – Resultados obtidos pelos vários algoritmos de pesquisa no mapa bigMaze

Como podemos verificar, em termos de tempo, é menos demorado do que o algoritmo UCS visto que pesquisa por menos níveis e em termos de nós expandidos é o algoritmo com o menor valor em ambas situações.

No próximo capítulo iremos apresentar o problema dos cantos e explicar a implementação desenvolvida.

Problema dos Cantos

O problema dos cantos consiste em encontrar o melhor caminho, consoante o algoritmo de pesquisa utilizado, de modo a passar por todos os pontos, localizados em cada canto do mapa. Nas próximas secções serão apresentadas a implementação desenvolvida e os resultados obtidos.

Implementação

A implementação da resolução ao *CornersProblem* começou pela criação de um novo estado abstrato. Este novo estado chamado `CustomState`[2], é caracterizado por conter unicamente a informação relativa à posição e uma estrutura do tipo *set* com os cantos previamente visitados. O objetivo inicial seria a utilização de uma lista para guardar informação

[2] https://github.com/pedrodlmatos/trabalhos-si/blob/main/desafio2/search/CustomState.py

relativa aos cantos visitados mas como, em python listas não são *hashable*, foi decidido fazer a adaptação para uma estrutura do tipo *set* de forma obter uma melhor performance.

```python
class CustomState:
    def __init__(self, position, visitedCorners):
        self.position = position
        self.visitedCorners = visitedCorners

    def __eq__(self, other):
        return isinstance(other, CustomState) \
               and self.position == other.position \
               and self.visitedCorners == other.visitedCorners

    def __hash__(self):
        return hash(self.position)

    def __str__(self):
        return f"(CustomState object) POSITION : {self.position} " \
               f"|| VISITED CORNERS : {self.visitedCorners}"
```

Figura 3 Implementação da classe `CustomState`

Após a definição do estado `CustomState`, foram definidos os métodos `getStartState()`, `isGoalState()` e `getSuccessors()`.

A lógica da implementação passa primeiramente pelo método `getStartState()` que retorna o estado inicial e um *set* vazio correspondente aos cantos previamente visitados. De seguida, o método `isGoalState()`, onde é definido o estado final do problema, sendo este uma verificação do número de cantos previamente visitados e se corresponde ao número de cantos existentes no mapa. Caso este seja o estado em que se encontra, o método retorna *True* e a pesquisa termina.

```
272  class CornersProblem(search.SearchProblem):
273      """This search problem finds paths through all four corners of a layout
278
279      def __init__(self, startingGameState, costFn = lambda x: 1):
280          """
281          Stores the walls, pacman's starting position and corners.
282          """
283          self.walls = startingGameState.getWalls()
284          self.startingPosition = startingGameState.getPacmanPosition()
285          top, right = self.walls.height-2, self.walls.width-2
286          self.corners = ((1,1), (1,top), (right, 1), (right, top))
287          for corner in self.corners:
288              if not startingGameState.hasFood(*corner):
289                  print(('Warning: no food in corner ' + str(corner)))
290          self._expanded = 0 # DO NOT CHANGE; Number of search nodes expanded
291          # Please add any code here which you would like to use
292          # in initializing the problem
293          "*** YOUR CODE HERE ***"
294          self.costFn = costFn
295
296      def getStartState(self):
297          """..."""
301          "*** YOUR CODE HERE ***"
302          return CustomState(self.startingPosition, set())
303
304      def isGoalState(self, state):
305          """..."""
308          "*** YOUR CODE HERE ***"
309          return len(state.visitedCorners) == len(self.corners)
```

Figura 4 Implementação das funções `getStartState()` e `isGoalState()` da classe `CornersProblem`

Por último, o método `getSuccessors()` que retorna os estados sucessores, as ações que eles exigem e um custo de 1 que é garantido pela variável `self.costFn` sendo atribuído por *default* uma função *lambda* de valor constante 1. Para cada estado é verificado se os seus sucessores não são paredes e se for esse o caso, é averiguado se algum dos sucessores é um canto a ser visitado e ainda é criada a lista de triplos contendo os estados sucessores, ações e o custo para mais tarde ser retornada pelo método.

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

    successors = []
    "*** YOUR CODE HERE ***"
    x, y = state.position

    for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)

        if not self.walls[nextx][nexty]:
            stateVisitedCorners = state.visitedCorners.copy()
            nextStatePosition = (nextx, nexty)
            if nextStatePosition in self.corners:
                stateVisitedCorners.add(nextStatePosition)

            cost = self.costFn(nextStatePosition)
            successors.append((CustomState(nextStatePosition, stateVisitedCorners), action, cost))

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

Figura 5 Implementação da função `getSucessors()` da classe `CornersProblem`

Resultados Obtidos

De forma a avaliar os resultados obtidos no problema dos cantos, como foi feito anteriormente, foi executado o agente nos diversos mapas utilizando os diversos algoritmos desenvolvidos até agora. Nas tabelas seguintes é possível avaliar os valores obtidos para cada um em termos de custo e nós expandidos para os *mediumMaze* e *bigMaze* respetivamente.

Algoritmo	**Custo**	**# Nós expandidos**	**Tempo (s)**
Depth-first search	221	371	0.0
Breadth-first search	106	1939	0.7
Uniform-cost search	106	1966	1.1
*A**	106	1966	0.7

Tabela 5 – Resultados obtidos pelos vários algoritmo de pesquisa no problema `CornersProblem` no mapa mediumMaze

Algoritmo	Custo	# Nós expandidos	Tempo (s)
Depth-first search	302	504	0.1
Breadth-first search	162	7945	11.5
Uniform-cost search	162	7949	16.0
*A**	162	7949	11.9

Tabela 6 – Resultados obtidos pelos vários algoritmo de pesquisa no problema `CornersProblem` *no mapa* bigMaze

Rapidamente é possível verificar a diferença do DFS para com os outros algoritmos. Apesar de oferecer uma resposta bastante rápida, vem com um custo significativamente maior. Naturalmente também é possível ver uma grande diferença entre pontuações dos outros algoritmos para com o DFS pois o DFS raramente oferece a solução ótima.

Havia também uma ideia de que o algoritmo *A** iria oferecer uma melhor performance em termos de nós expandidos para com o BFS e o UCS mas ao fazer a comparação de resultados, verificámos o contrário. Após refletir sobre os resultados chegámos à conclusão que isto simplesmente era devido à falta da função heurística implementada, heurística essa que será implementada no próximo step. Mesmo assim é possível verificar que para o caso, existe uma superioridade do BFS e *A** sobre o UCS.

É de notar que o A*, tal como se encontra referenciado no enunciado, tem um custo total de 28 passos para completar o *tinyCorners*.

```
[SearchAgent] using function astar and heuristic nullHeuristic
[SearchAgent] using problem type CornersProblem
Path found with total cost of 28 in 0.0 seconds
Search nodes expanded: 252
Pacman emerges victorious! Score: 512
Average Score: 512.0
Scores:        512.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Figura 6 Output *obtido utilizando o algoritmo* A* *sem função heurística no mapa* mediumMaze

A implementação do *CornersProblem*, tal como se encontra no relatório, passa por resolver o problema mesmo que só um dos cantos do mapa tenha comida. Existe só o pequeno pormenor

que caso o Pac-Man coma a única comida existente no mapa, o jogo acaba e portanto não deixa o seu *pathing* continuar.

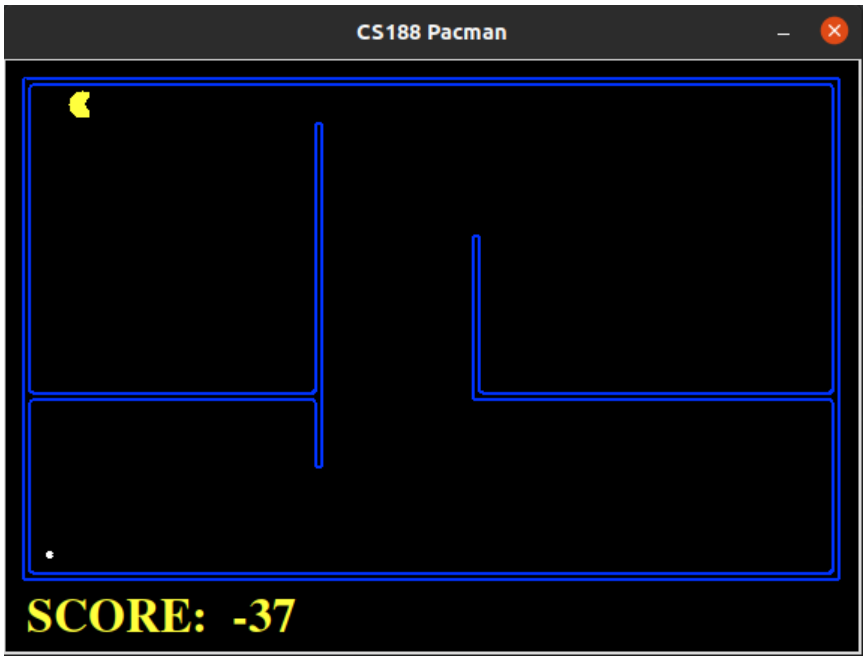

Figura 7 Mapa openMaze *no problema* CornersProblem

```
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Warning: no food in corner (1, 21)
Warning: no food in corner (35, 1)
Warning: no food in corner (35, 21)
Path found with total cost of 116 in 16.7 seconds
Search nodes expanded: 7335
Pacman emerges victorious! Score: 428
Average Score: 428.0
Scores:        428.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Figura 8 Output *obtido quando se executa o agente num mapa sem os cantos todos ocupados*

Execução

A classe *main* recebe alguns parâmetros que definem o ambiente onde decorre o jogo. Para esta etapa do desafio, os argumentos mais importantes e os respetivos valores que podem tomar são:

- Mapa: `-l` ou `--layout`
  - `tinyMaze`
  - `smallMaze`
  - `mediumMaze`
  - `bigMaze`
  - `openMaze`
  - `tinyCorners`
  - `mediumCorners`
  - `bigCorners`
- Agente inteligente: `-p` ou `--pacman`
  - `SearchAgent`
- Argumentos do agente: `-a` ou `--agentArgs`
  - `fn=astar`
  - `fn=astar,heuristic=manhattanHeuristic`
  - `fn=astar,heuristic=euclideanHeuristic`
  - `fn=bfs,prob=CornersProblem`

Desse modo, para testar o agente inteligente no mapa `mediumMaze` utilizando o algoritmo A* com a função heurística *manhattan*, o comando a ser executado é:

- `python3 pacman.py -l mediumMaze -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic`

O resultado obtido para este comando está presente na seguinte figura.

```
[SearchAgent] using function astar and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 221
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
```

Figura 9 Output *do comando descrito*

Quando os comandos são executados. é aberto um mapa onde mostra os agente a movimentar-se e os nós expandidos. A figura abaixo apresenta o mapa obtido para o comando executado acima.

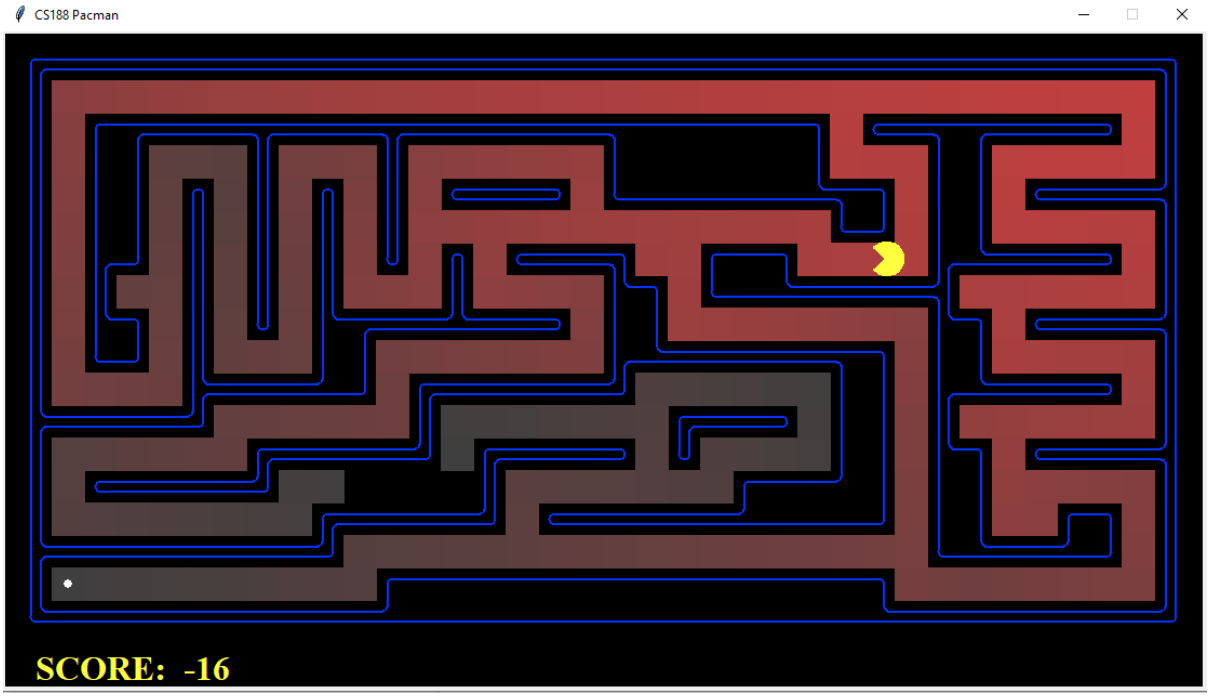

Figura 10 Exemplo de execução do Pac-man

Para executar o problema CornersProblem o comando a ser executado é diferente. Para melhor perceber o foco do problema, é preferível utilizar um dos mapas próprios para o problema. Assim, para executar o agente no mapa *mediumCorners* utilizando o algoritmo BFS, o comando a ser executado é:

- `python3 pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem`

```
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 106 in 2.6 seconds
Search nodes expanded: 1936
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Figura 11 Output do comando descrito

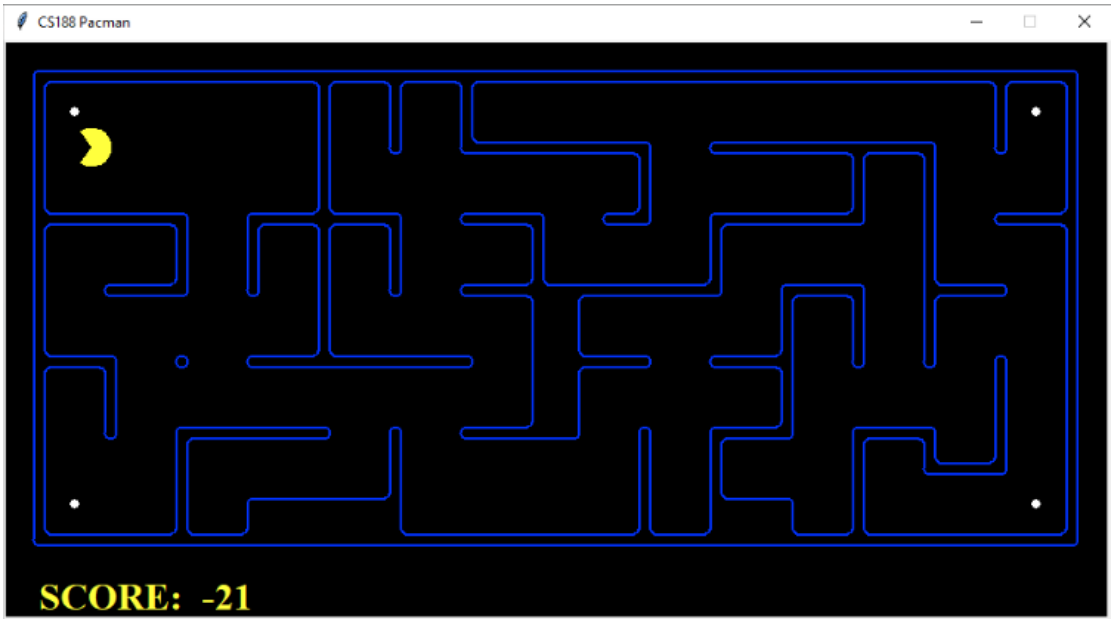

Figura 12 Exemplo de execução do Pac-man no problema CornersProblem

Todo o código fonte bem como instruções mais detalhadas para executar o projeto podem ser encontradas no repositório[3] no Github.

Conclusão

Estas etapas deste desafio foram importantes para poder comparar os resultados para cada algoritmo desenvolvido e para contrastar quais as diferenças entre eles. Para além disso, foi interessante perceber as diferenças teóricas entre cada um num caso prático com resultados reais e possíveis de serem interpretados.

No entanto, concordamos que os resultados obtidos no problema *CornersProblem* não vão de encontro aos que se esperavam visto que em situações normais, o algoritmo A* apresentava melhores resultados que os restante mas neste caso isso não se verifica.

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