universidade de aveiro deti departamento de electrónica, telecomunicações e informática

# Sistemas Inteligentes

2020/2021

# Desafio 2, Step 2 e 3 - Breadth-First e Uniform-Cost search


Pedro Matos

84986

pedrolopesmatos@ua.pt

Simão Arrais

85132

simaoarrais@ua.pt


maio de 2021

# Conteúdo



# 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) ou *Uniform-cost search* (UCS) 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 segunda e terceira etapas deste desafio consistem no desenvolvimento dos algoritmos BFS e UCS e nos próximos capítulos iremos detalhar o funcionamento destes algoritmos bem como a sua implementação no jogo do Pac-man.

# Breadth-First Search

O algoritmo BFS é um outro exemplo de um algoritmo de pesquisa não informado que opera em força bruta de forma a chegar ao objetivo. A estratégia passa por expandir e explorar todos os nós de um nível antes de passar para o nível seguinte, ou seja, ao expandir o nó inicial,

primeiramente vão ser expandidos todos os nós sucessores ao nó raiz e só depois é que se procura a expansão dos seus sucessores do nível seguinte. Apesar de usar força bruta, o BFS pode ser usado para encontrar o caminho mais curto de origem única num grafo sem custos pois o BFS alcança um vértice com um número mínimo de arestas.

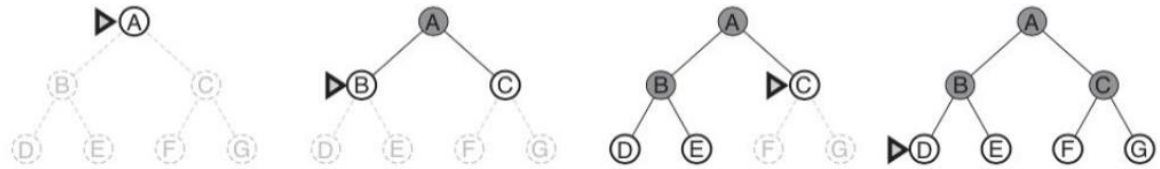


*Figura 1 Progresso da pesquisa numa árvore binária*

# Implementação

A implementação do BFS passa por utilizar uma Queue[1] como estrutura de dados para guardar os nós a visitar e uma lista para guardar os nós já visitados. A Queue é uma estrutura de dados que segue a logística de armazenamento FIFO (First-In First-Out), ou seja, o primeiro nó a ser adicionado é o primeiro a ser expandido. Tal como o DFS, o algoritmo só consegue distinguir entre estado normal e o estado alvo, portanto, começa por verificar se o estado inicial ou estado a ser averiguado é o estado alvo e, se tal se verifica, retorna uma lista vazia e termina. Caso contrário, é adicionado o estado a ser averiguado à Queue e até ser encontrado o estado alvo ou quando já não existirem nós sucessores, é feita a pesquisa. Os nós não são adicionados à Queue caso já tenham sido visitados de forma a evitar a criação de ciclos.

[1] ficheiro `util.py`

```python
def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"
    actions = Queue()  # stack structure (from utils)
    visited_states = []  # visited states

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

    # Push initial state and path to it (empty path)
    initial_node = (problem.getStartState(), [])
    actions.push(initial_node)

    # Stop when solution is not found
    while not actions.isEmpty():
        # get current state and path to it
        current_state, path_to_state = actions.pop()
        # print(path_to_state)

        # verify if state is goal state
        if problem.isGoalState(current_state):
            return path_to_state

        if current_state not in visited_states:
            # add state in list of visited states
            visited_states.append(current_state)

            # Get child nodes of current state
            child_states = problem.getSuccessors(current_state)
            #shuffle(child_states)

            if len(child_states) > 0:
                # add child nodes to list of actions
                for node in child_states:
                    state, direction, cost = node
                    if state not in visited_states:
                        child_path = path_to_state + [direction]
                        actions.push((state, child_path))

    return []
```

*Figura 2 Progresso da pesquisa numa árvore binária*

# Resultados Obtidos

Para avaliar os resultados obtidos, executamos o agente nos diversos mapas e 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 | 0.0 |

Tabela 1 - Resultados de execução do BFS

É possível verificar facilmente a metodologia do BFS através da figura 3 onde podemos ver a exploração de quase todos os nós. Os nós que acabaram por não ser explorados, fariam parte de um nível seguinte de nós vizinhos.

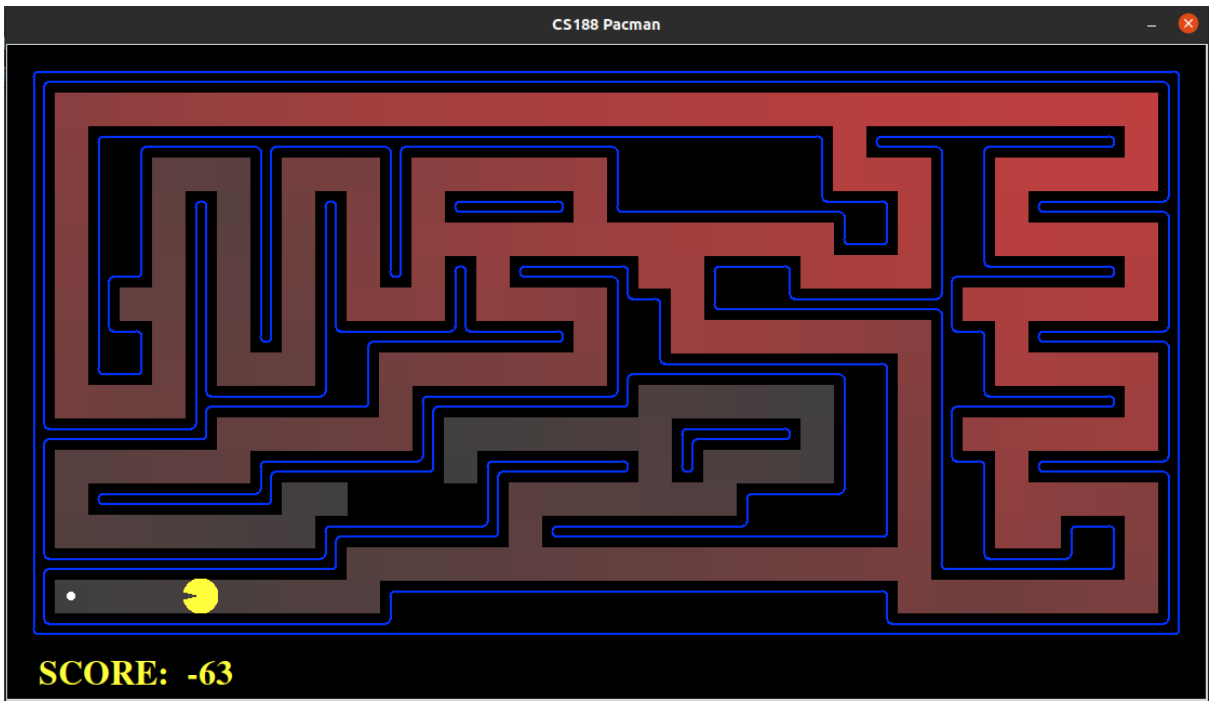


*Figura 3 Exemplo de execução com BFS num mediumMaze*

Como o algoritmo considera primeiro todos os vizinhos disponíveis, é de esperar que não seja o algoritmo mais eficiente para árvores onde seja necessário tomar decisões ou que tenham um custo associado, mas, como foi testado num ambiente onde só é necessário que o Pac-man se desloque de A a B, com a sua metodologia de força bruta e análise de todos os vizinhos disponíveis, oferece sempre uma solução ideal.

É ainda de notar que a implementação do algoritmo BFS foi feita de forma genérica tanto que é possível obter resultados correndo o jogo 8 Puzzle.

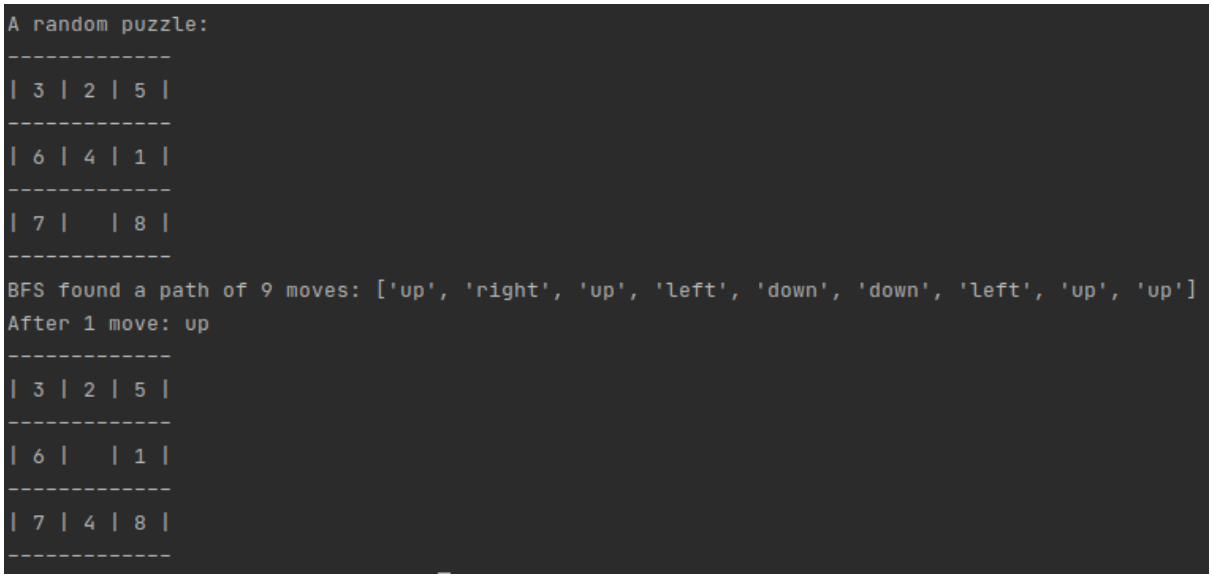


*Figura 4 Exemplo de execução com BFS no 8 Puzzle*

O comando para executar o 8 Puzzle é:

- `python3 eightpuzzle.py`

# Uniform-Cost Search

O algoritmo UCS é um algoritmo de pesquisa não informado que, ao contrário dos algoritmos estudados anteriormente, tem em consideração o custo para alcançar cada nó, ou seja, para cada nó que está a ser explorado, é expandido pelo nó de menor custo. Para guardar os estados a explorar, é utilizada uma Priority Queue que é ordenada de forma crescente pelo

valor do custo de cada nó guardado (*frontier*) e, também, uma lista para guardar os nós já visitados.

Na próxima secção iremos apresentar e discutir a implementação deste algoritmo no jogo do Pac-man.

# Implementação

O código fornecido já tinha desenvolvido uma Priority Queue[2] que foi utilizada na implementação do algoritmo, denominando-a *frontier*, para guardar os estados a visitar. Os dados são guardados na estrutura como um tuplo: o primeiro elemento é 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 *frontier* 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 explorados. Por cada vez que o ciclo se repete, é extraído da *frontier* 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, adiciona-se o estado à lista de estados visitados. De seguida, para cada nó descendente, se ainda não estiver na lista de estados visitados ou na *frontier* , então é adicionado à última; caso já se encontre na *frontier* e com um custo maior, então o valor do custo é atualizado.

Em último lugar, e caso não seja encontrada uma solução, o algoritmo retorna uma lista vazia.

Na próxima figura está apresentada a implementação deste algoritmo.

[2] ficheiro `util.py`

```python
def uniformCostSearch(problem):
    """Search the node of least total cost first."""

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

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

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

    # stop when solution is found or when queue is empty
    while not frontier.isEmpty():
        current_state, path_to_state = frontier.pop()  # choose the lowest cost node

        if problem.isGoalState(current_state):
            return path_to_state

        if current_state not in visited_states:
            # add current state in visited states
            visited_states.append(current_state)

            # get child nodes
            child_states = problem.getSuccessors(current_state)

            if len(child_states) > 0:
                # add child nodes to list of actions
                for node in child_states:
                    state, direction, cost = node
                    path_to_child = path_to_state + [direction]
                    path_cost = problem.getCostOfActions(path_to_child)
                    frontier_states = {state: priority for priority, _, (state, directions) in frontier.heap}

                    if state not in visited_states or state not in frontier_states:
                        frontier.push((state, path_to_child), path_cost)
                    elif state in frontier_states and frontier_states[state] > path_cost:
                        frontier.update((state, path_to_child), path_cost)
    return []
```

*Figura 5 Código da implementação do algoritmo Uniform-cost search*

Na próxima secção, serão apresentados os resultados obtidos para o algoritmo UCS.

## Resultados Obtidos

Para avaliar os resultados obtidos, executamos o agente nos diversos mapas e observamos os valores obtidos para o custo, para o número de nós expandidos e para o tempo

que o agente demorou a explorar. Os resultados obtidos para o algoritmo desenvolvido estão presentes na tabela abaixo.

| Mapa | Custo | # Nós expandidos | Tempo (s) |
|---|---|---|---|
| *tinyMaze* | 8 | 15 | 0 |
| *smallMaze* | 19 | 92 | 0 |
| *mediumMaze* | 68 | 269 | 0.1 |
| *bigMaze* | 210 | 620 | 0.3 |

*Tabela 2 Resultados obtidos usando o algoritmo* Uniform-cost search

O número de nós explorados é bastante elevado, o que resulta em termos práticos, num aumento do tempo que demora a explorar todos os nós.

Para além disso, avaliamos também a performance dos agentes já desenvolvidos, como o *StayEastSearchAgent* e *StayWestSearchAgent*. Estes dois agentes, devido às suas funções de custo exponenciais, apresentam resultados muito baixos e muito elevados, respetivamente. A tabela abaixo mostra os valores obtidos usando estes dois agentes nos diversos mapas.

| Agente | Mapa | Custo | # Nós expandidos | Tempo (s) |
|---|---|---|---|---|
| *StayEastSearchAgent* | *tinyMaze* | 1 | 14 | 0 |
| | *smallMaze* | 1 | 85 | 0 |
| | *mediumMaze* | 1 | 260 | 0.1 |
| | *mediumDottedMaze* | 1 | 186 | 0.0 |
| | *bigMaze* | 5 | 639 | 0.3 |
| *StayWestSearchAgent* | *tinyMaze* | 48 | 11 | 0 |
| | *smallMaze* | 23806 | 58 | 0 |
| | *mediumMaze* | $\approx 1{,}72 \times 10^{10}$ | 173 | 0.1 |
| | *mediumScaryMaze* | $\approx 6{,}87 \times 10^{10}$ | 108 | 0.0 |
| | *bigMaze* | $\approx 5{,}9 \times 10^{11}$ | 497 | 0.2 |

*Tabela 3 Resultados obtidos usando os agentes já desenvolvidos nos diferentes mapas*

Os mapas *mediumDottedMaze* e *mediumScaryMaze* têm as suas funções de custo alteradas o que leva a que os resultados obtidos, comparativamente ao mapa *mediumMaze*, sejam melhores, provando assim que alterando essas funções, os agentes podem seguir percursos melhores e adaptar-se a diferentes situações.

Tendo em conta que já foram desenvolvidos alguns algoritmos de pesquisa, é-nos agora possível fazer a comparação dos respetivos resultados. Nas próximas tabelas estão presentes os resultados obtidos nos mapas mediumMaze e bigMaze usando os diversos algoritmos desenvolvidos até ao momento.

| **Algoritmo** | **Custo** | **# Nós expandidos** | **Tempo (s)** |
|---|---|---|---|
| *Depth-first search* | 130 | 146 | 0 |
| *Breadth-first search* | 68 | 268 | 0 |
| *Uniform-cost search* | 68 | 269 | 0 |

*Tabela 4 Resultados obtidos usando os vários algoritmos de pesquisa no mapa* mediumMaze

| **Algoritmo** | **Custo** | **# Nós expandidos** | **Tempo (s)** |
|---|---|---|---|
| *Depth-first search* | 210 | 390 | 0 |
| *Breadth-first search* | 210 | 622 | 0.1 |
| *Uniform-cost search* | 210 | 620 | 0.2 |

*Tabela 5 Resultados obtidos usando os vários algoritmos de pesquisa no mapa* bigMaze

Tal como o algoritmo DFS, estes agora desenvolvidos são também não informados o que significa que toda a informação adicional que poderia ajudar a obter melhores resultados, como heurísticas, não é considerada.

O algoritmo DFS, no mapa *mediumMaze*, apresenta um custo muito superior que os restantes algoritmos e isso deve-se à sua pesquisa em profundidade visto que pesquisa mais níveis e, por isso, aumenta o custo mesmo conseguindo encontrar uma solução sem explorar tantos nós. Seguindo outras formas de explorar, é possível obter um caminho com menor custo apesar de ter um número de nós expandidos maior.

Tendo em conta que o custo entre cada nó é de 1, os resultados obtidos para os algoritmos BFS e UCS são bastante semelhantes, mas em situações em que isso não se verificasse, o algoritmo UCS, possivelmente, destacar-se-ia.

# 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`
  - `mediumDottedMaze`
  - `mediumScaryMaze`
- Agente inteligente: `-p` ou `--pacman`
  - `SearchAgent`
  - `StayWestSearchAgent`
  - `StayEastSearchAgent`
- Argumentos do agente: `-a` ou `--agentArgs`
  - `fn=bfs`
  - `fn=ucs`

Desse modo, para testar o agente inteligente no mapa `mediumMaze` utilizando o algoritmo UCS, o comando a ser executado é:

- `python3 pacman.py -l mediumMaze -p SearchAgent -a fn=ucs`

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

```
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
```

*Figura 6* Output *obtido no terminal*

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.

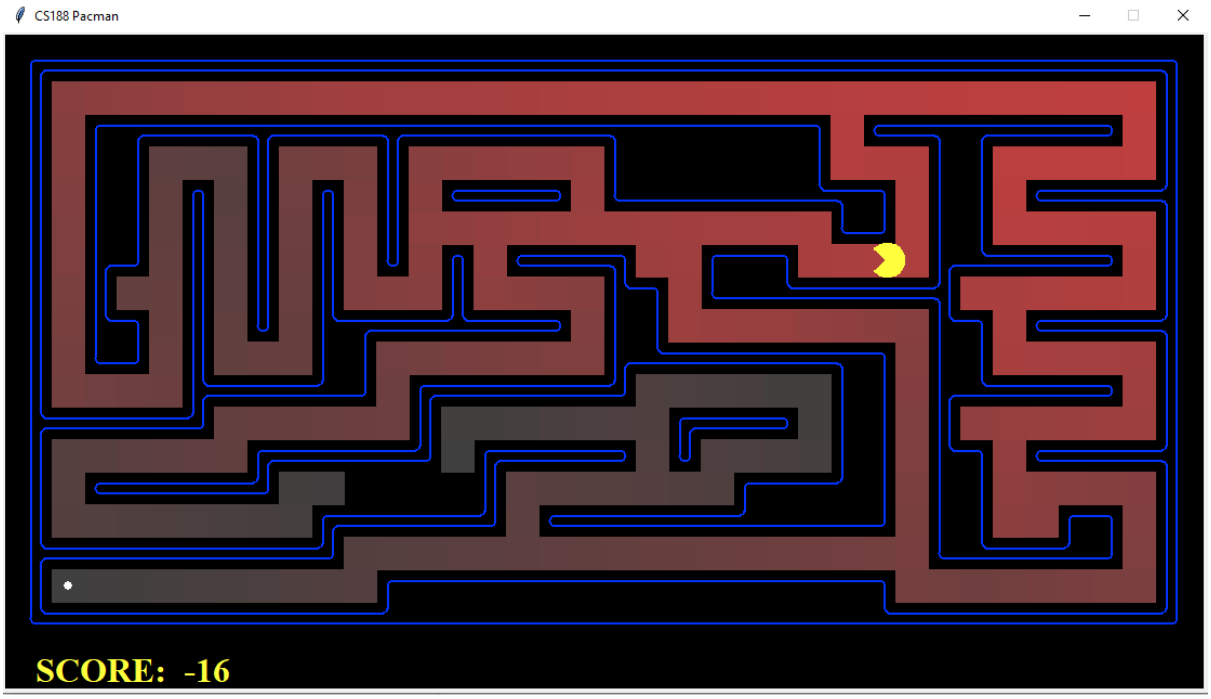


*Figura 7* mediumMaze *mostrado quando se executa a intrução*

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

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

# 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.