

Sistemas Inteligentes

2020/2021

Desafio 2, Step 1 - Depth-first search

Pedro Matos

84986

pedrolopesmatos@ua.pt

Simão Arrais

85132

simaoarraais@ua.pt

maio de 2021

Conteúdos

Introdução	2
Depth-first Search	2
Implementação	3
Resultados obtidos	5
Execução	6
Conclusão	8

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.

A primeira etapa deste desafio consiste no desenvolvimento do algoritmo DFS e nos próximos capítulos iremos detalhar o funcionamento deste algoritmo e a sua implementação no jogo do Pac-man.

Depth-first Search

O algoritmo DFS é um algoritmo de pesquisa não informado em profundidade utilizado para pesquisar em estruturas de árvore ou grafo e o seu funcionamento consiste em, a partir de um nó inicial, expandir todos os descendentes desse nó, e assim sucessivamente, até a atingir um nó sem descendentes ou até encontrar o objetivo. No caso de atingir um nó

sem descendentes, então o algoritmo retrocede ao “pai” desse nó e segue a pesquisa por outro dos nós descendentes.

A estrutura de dados utilizado neste algoritmo é uma LIFO (Last-In First-Out), ou seja, o último nó a ser adicionado é o primeiro a ser expandido. A figura 1 mostra o funcionamento deste algoritmo apresentando a informação guardada na LIFO em cada fase.

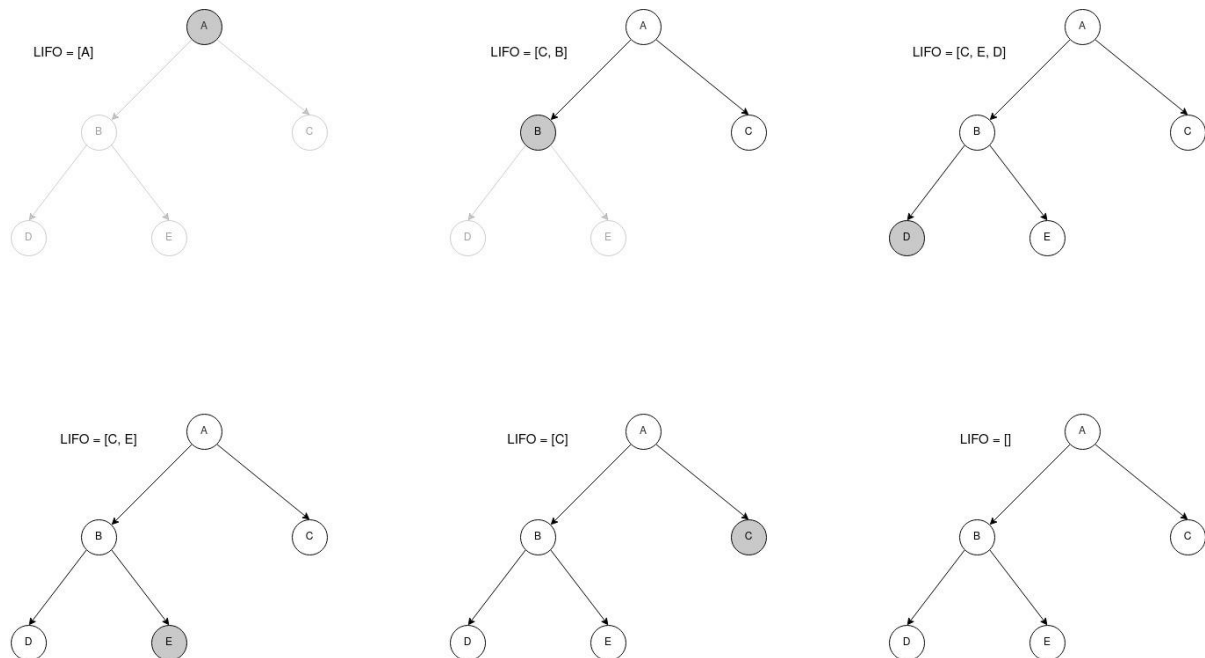


Figura 1 Esquema representativo do funcionamento do algoritmo Depth-first search

Tendo em conta que este algoritmo não é informado, toda a informação adicional que poderia ajudar a obter melhores resultados, por exemplo, a utilização de heurísticas, não é aplicável.

Na próxima secção iremos detalhar a implementação do algoritmo aplicada ao jogo do Pac-man.

Implementação

Tendo em conta a simplicidade deste algoritmo e a existência de estruturas já desenvolvidas, a implementação deste algoritmo foi bastante simples. No caso deste desafio,

o objetivo é encontrar o estado alvo e o que é retornado é uma lista com as direções seguidas em cada passo.

```
79 def depthFirstSearch(problem):
80     """
81     Search the deepest nodes in the search tree first.
82
83     Your search algorithm needs to return a list of actions that reaches the
84     goal. Make sure to implement a graph search algorithm.
85     """
86     actions = Stack()          # stack structure (from utils)
87     visited_states = []        # visited states
88
89     # if initial state is the goal state (stop)
90     if problem.isGoalState(problem.getStartState()):
91         return []
92
93     # Push initial state and path to it (empty path)
94     actions.push((problem.getStartState(), []))
95
96     # Stop when solution is found
97     while not actions.isEmpty():
98         # get current state and path to it
99         current_state, path_to_state = actions.pop()
100
101         # verify if state is goal state
102         if problem.isGoalState(current_state):
103             return path_to_state
104
105         if current_state not in visited_states:
106             # add state in list of visited states
107             visited_states.append(current_state)
108
109             # Get child nodes of current state
110             child_states = problem.getSuccessors(current_state)
111             #shuffle(child_states)
112
113             if len(child_states) > 0:
114                 # add child nodes to list of actions
115                 for node in child_states:
116                     state, direction, cost = node
117                     if state not in visited_states:
118                         child_path = path_to_state + [direction]
119                         actions.push((state, child_path))
120
121     return []
```

Figura 2 Implementação do algoritmo Depth-first search

Para guardar os estados a visitar, é utilizado uma *stack*¹ e para guardar os estados visitados é utilizada uma lista. Para começar é averiguado se o estado inicial é o estado alvo e, se sim, retorna uma lista vazia e termina. Caso contrário, é adicionado à *stack* o estado inicial e enquanto não for encontrado o estado alvo ou quando já não existirem nós por explorar, é feita a pesquisa e os nós são adicionados à *stack* caso ainda não tenham sido visitados, para evitar a criação de ciclos.

Tendo em conta que o algoritmo não é ótimo, decidimos estudar a influência que a ordem com que os descendentes de um nó são visitados tem. Para isso, utilizamos a função `shuffle` do módulo `random` (linha 111).

Na próxima secção iremos discutir os resultados obtidos analisando o impacto que a ordem com que os nós descendentes são explorados tem no resultado final.

Resultados obtidos

O desafio, após cada tentativa, retorna o custo do trajeto escolhido pelo agente, ou seja, o tamanho do percurso percorrido, e o principal objetivo é minimizar o custo escolhendo os melhores caminhos. Na tabela seguinte está presente o custo obtido numa tentativa para cada mapa e avaliando a ordem com que os descendentes de um nó são explorados.

Mapa	Ordem normal	Ordem aleatória
<i>tinyMaze</i>	10	10
<i>smallMaze</i>	49	37
<i>mediumMaze</i>	130	144
<i>bigMaze</i>	210	210

Tabela 1 – Resultados obtidos numa tentativa

Como podemos verificar, nalguns casos o custo da solução obtida usando uma ordem aleatória é melhor que usando a ordem normal. No entanto, para clarificar a influência, decidimos avaliar tendo como base 100 tentativas.

¹ desenvolvida no ficheiro `util.py`

Mapa	Ordem normal	Ordem aleatória
<i>tinyMaze</i>	10	9
<i>smallMaze</i>	49	34
<i>mediumMaze</i>	130	118
<i>bigMaze</i>	210	210

Tabela 2 – Média dos resultados obtidos em 100 tentativas

Como seria de esperar, o custo quando a ordem é normal mantém-se porque a ordem não se altera em diferentes execuções e, para além disso, podemos verificar que, a ordem, de facto, influencia o resultado obtido: de uma forma geral, a ordem normal com que os nós são explorados não é a que obtém o melhor percurso.

Na próxima secção, iremos mostrar como executar o código e o output obtido.

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`
- Agente inteligente: `-p` ou `--pacman`
 - `SearchAgent`
- Argumentos do agente: `-a` ou `--agentArgs`
 - `fn=dfs`

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

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

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

```
[SearchAgent] using function dfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 130 in 0.0 seconds
Search nodes expanded: 146
Pacman emerges victorious! Score: 380
Average Score: 380.0
Scores:      380.0
Win Rate:    1/1 (1.00)
Record:      Win
```

Figura 3 Output obtido executando o agente no mapa medimMaze

Quando os comandos são executados, uma janela é aberta onde se pode ver o agente a movimentar-se pelo mapa.

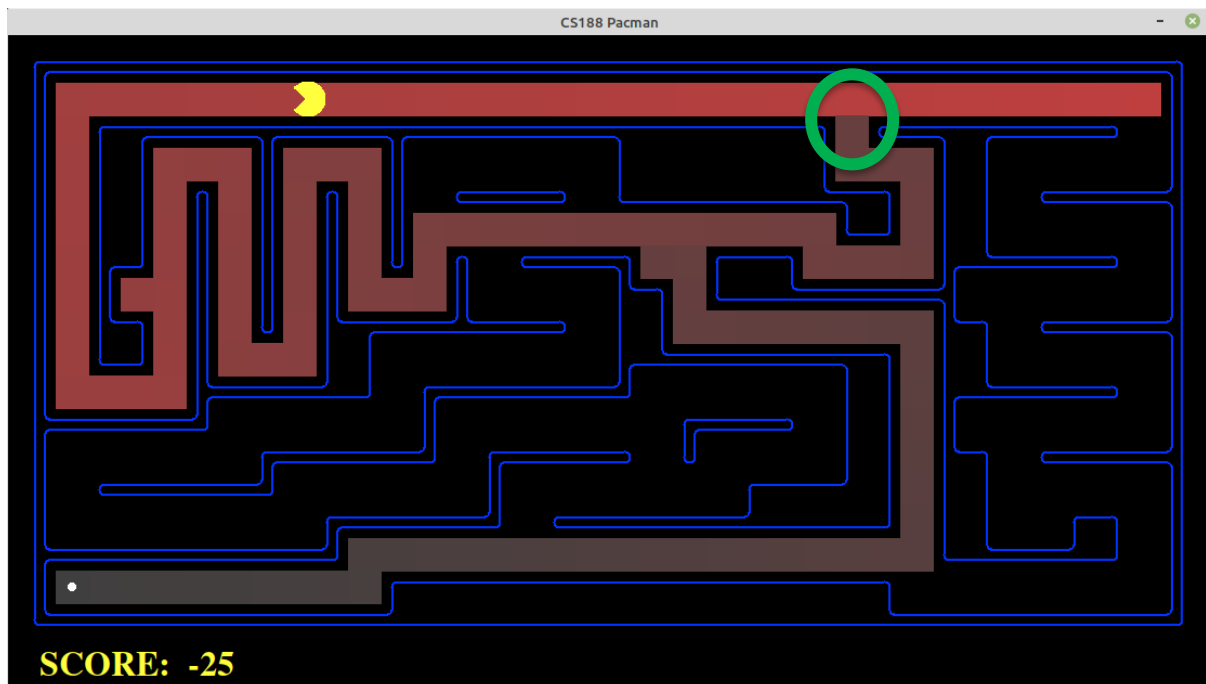


Figura 4 Mapa mediumMaze

Este mapa contém uma traçado que representa os estados que são removidos da stack e respetiva ordem, quanto mais forte a cor, mais cedo foram removidos da stack. Existem quadrados que aparecem explorados mas pelos quais o agente não passa visto que consegue encontrar uma solução passando por outros estados. Visualmente, isso pode representar uma diferença repentina na cor do traçado no mapa (zona assinalada a verde na figura).

Como podemos observar na imagem, apesar de a ordem com que o mapa é explorado parecer correta, este algoritmo não é ótimo visto que o caminho que se seguiu é mais longo do que se tivesse seguido a direção SOUTH na zona marcada na figura, por exemplo.

Para facilitar, foi criado um ficheiro *Makefile* para executar os comandos de forma mais intuitiva. Como nesta fase do desafio, o parâmetro mais variável é o mapa, foi criado uma etapa no ficheiro para cada mapa. Assim, para executar o ambiente anterior basta correr o comando `make mediumMaze` no terminal.

Todo o código fonte está guardado no repositório² no Github e todas as instruções necessárias estão contidas no ficheiro README.md.

Conclusão

Esta etapa deste desafio foi importante para fazer a introdução ao temas dos algoritmos de pesquisa e para perceber quais as suas vantagens e desvantagens e em que situações é devem ser utilizados.

No caso do algoritmo DFS conseguimos perceber as consequências que advêm do facto de não ser ótimo, nomeadamente nos resultados. Para além disso, percebemos também que utilizar a ordem normal em vez de explorar ordens alternativas pode resultar também em resultados piores.

² <https://github.com/pedrodlmatos/trabalhos-si/tree/main/desafio2>