universidade de aveiro deti departamento de electrónica, telecomunicações e informática

Sistemas Inteligentes

2020/2021

Desafio 2, Step 1 - Depth-first search

Pedro Matos

84986

pedrolopesmatos@ua.pt

Simão Arrais

85132

simaoarrais@ua.pt

maio de 2021

Conteúdos

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* ou *Uniform-cost search* 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 do mesmo 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.

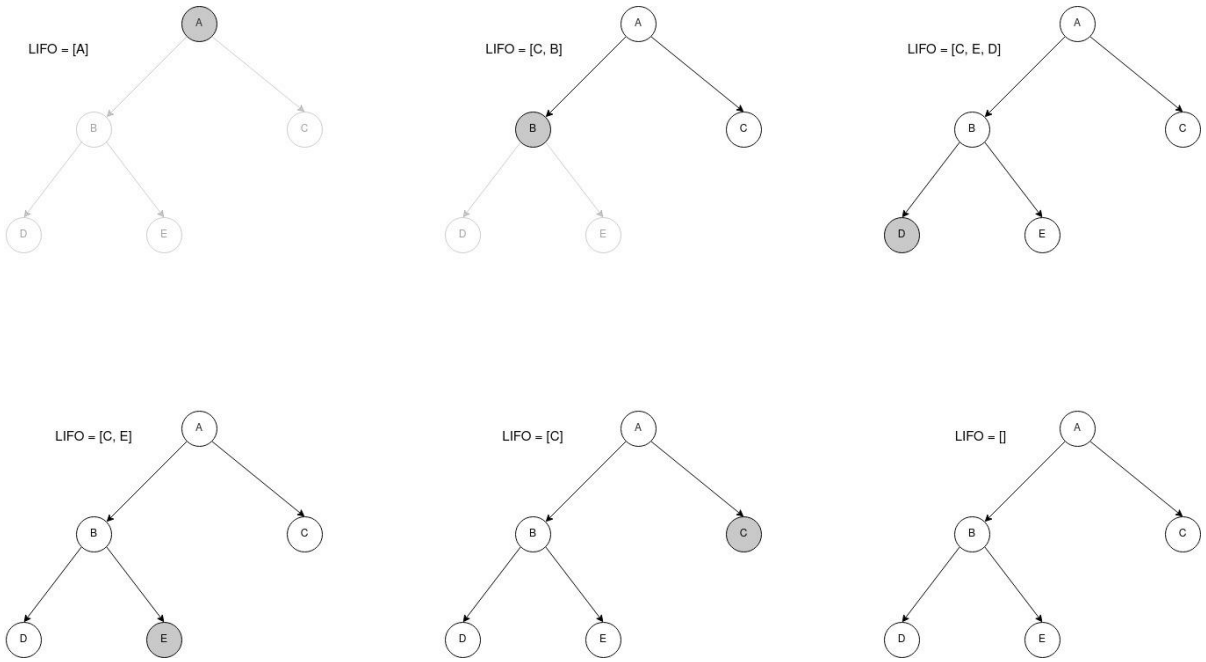

Figura 1 Esquema representativo do funcionamento do algoritmo Depth-first search

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.

```python
def depthFirstSearch(problem):
    """
    Search the deepest nodes in the search tree first.

    Your search algorithm needs to return a list of actions that reaches the
    goal. Make sure to implement a graph search algorithm.
    """
    actions = Stack()                   # 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)
    actions.push((problem.getStartState(), []))

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

        # 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 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**
tinyMaze	10	10
smallMaze	49	37
mediumMaze	130	144
bigMaze	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
tinyMaze	10	9
smallMaze	49	34
mediumMaze	130	118
bigMaze	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 mediumMaze

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

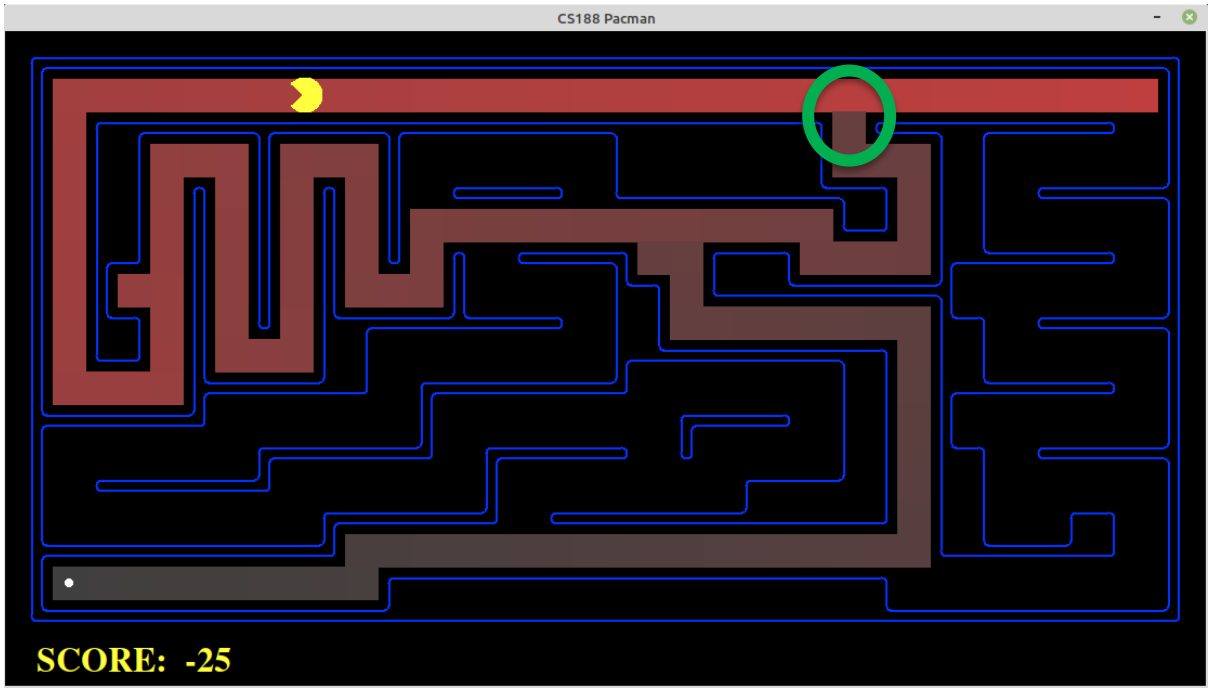

Figura 4 Mapa mediumMaze

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, sendo que a *flag* -n não funcionou corretamente, 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 repositório[2] 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