Relatório

Elementos do grupo

- Gelson Stalino Gomes Varela – nº 202109347
- João Filipe Pacheco Vivas Vivas - n° 202108177
- Tomás Martins - n° 201506226

1. Introdução

Um problema de busca é algo que pode ser resolvido partindo de um **estado inicial** e atravessando um **conjunto de estados** com o objetivo de atingir um ou mais objetivos ou **estados finais**. Essa mudança de estado é chamada de ação e só pode haver um conjunto de ações a partir de um estado *s*. As ações executadas criam um resultado, esse resultado é descrito pelo **modelo de transição**. Cada ação também tem um custo ao ser executada que vai servir para medir a performance do programa para atingir o objetivo.

Existem várias maneiras de resolver estes problemas. Os métodos usados podem ser agrupados em estratégias de busca não guiada (“cega”) ou guiada (informada).

As estratégias de busca não guiada são a **pesquisa em largura**, **pesquisa em profundidade, pesquisa em profundidade iterativa limitada**, **pesquisa de busca de custo uniforme** ou **pesquisa de busca bidirecional**. As estratégias de busca guiada são, por exemplo, a pesquisa **“gulosa”** com heurística e **A*** com heurística.

2. Descrição do problema

O problema é o jogo dos 15. Este jogo consiste num tabuleiro constituído por uma grelha de 4x4 onde cada célula tem um símbolo atribuído, exceto uma que está vazia. O objetivo é tentar colocar o tabuleiro que se encontra numa configuração inicial para uma configuração final também ela definida a priori. Podendo apenas deslizar as peças adjacentes para a posição da célula vazia sendo que esse movimento só pode ser feito na vertical ou na horizontal. Este problema só tem um certo número de configurações que podem atingir o objetivo através de movimentos legais, sendo esse número $\frac{n!}{2}$ sendo *n* o número de peças no tabuleiro.

3. Estratégias de busca

a) Busca não guiada

- **Profundidade (*Depth-First Search, DFS*)**: Consiste em partir de um nó inicial e verificar se esse nó é solução. Caso não seja, seleciona-se um nó descendente e verifica-se se é solução, se não for repete-se o processo sucessivamente. Só se muda de nó descendente caso se tenha a certeza que nenhuma sub-árvore do nó descendente anterior der para o objetivo. Esta estratégia é usada quando

uma pesquisa em profundidade é a mais indicada devido ao seu baixo uso de memória, apesar que pode não encontrar a solução ótima. A complexidade temporal é $O(b^m)$ e a complexidade espacial é $O(bm)$onde *b* é o fator de ramificação (número de filhos por nó) e o *m* é a máxima profundidade da árvore.

- **Largura (*Breadth-First Search, BFS*)**: Consiste em partir de um nó inicial e verificar se é a solução, caso não seja verifica se cada filho é a solução. Se nenhum desses nós for a solução, todos os nós são expandidos e verifica-se se há algum nó no nível de seguinte que é a solução e assim sucessivamente. Esta estratégia é boa se quisermos a solução ótima, mas para resultados que geram árvores muito complexas é pior, pois precisa de guardar todos os nós até encontrar a solução o que não é muito bom a nível de memória. A complexidade temporal é$O(b^d)$e a complexidade espacial é$O(b^d)$onde *b* é o fator de ramificação e *d* é a profundidade da solução mais funda.
- **Busca Iterativa Limitada em Profundidade (*Iteractive Depth-First Search, IDFS*)**: Parte de um nó inicial e faz uma procura em profundidade, mas só procura até a uma certa profundidade. Se a solução não for encontrada repete-se o processo, mas aumentasse o limite da procura até ao nível de profundidade seguinte. Com esta estratégia podemos encontrar a solução ótima e temos uma utilização de memória menor que a pesquisa em largura. A complexidade temporal é $O(b^l)$ e a complexidade espacial $O(bl)$ onde *b* é o fator de ramificação e *l* é a profundidade do limite.

b) Busca guiada

Estas pesquisas também chamadas pesquisas informadas precisam de uma **heurística**. Uma heurística é uma estimativa da distância que falta para atingir o objetivo. Neste problema nós usamos duas. O número de peças fora do sítio e o somatório da *Manhattan distance* (distância para o destino usando linhas retas verticais e horizontais).

- **Pesquisa gulosa (*greedy search*)**: Esta estratégia verifica se o nó inicial é a solução e se não for seleciona o descendente que tem a menor heurística, ou seja, que é mais próximo de encontrar a solução, e assim sucessivamente. Este método é incompleto e não é ótimo.
- **A***. Esta estratégia verifica se o nó inicial é a solução, se não for seleciona o nó descendente com base nesta fórmula **f(n) = g(n) + h(n)**. Onde **f(n)** é o custo estimado do melhor caminho que passa pelo nó *n* até o objetivo. **g(n)** é o custo do caminho da raiz até ao nó *n* e **h(n)** é o custo estimado para chegar até ao final. Esta estratégia permite encontrar a solução ótima se a heurística for admissível (que não sobrestima o custo real da melhor solução).

4. Implementação

Decidimos usar **Python** para implementar as estratégias de busca. Devido à sua sintaxe simples e ótima legibilidade, e também pelo fato de ser possível criar classes de dados personalizados, irá nos permitir fazer uma boa organização do código e escrever

programas reutilizáveis que podem ser úteis para implementar as diferentes estratégias de busca.

Estruturas de dados

Criamos uma estrutura de dados, ***node*** que guarda os seguintes valores como atributos:

- **Estado**: vai guardar uma lista que representa as posições das peças no tabuleiro daquele nó;
- **Parent**: vai indicar qual o nó gerou o nó atual, se o nó atual for a raíz vai ter um valor de *None*;
- **MoveSet**: guarda na forma de string quais os movimentos que foram feitos para se atingir aquele nó (L – mover a peça da esquerda; R – mover a peça da direita; U – mover a peça de cima; D – mover a peça de baixo);
- **BlankPos**: guarda a posição na lista do estado em que a peça vazia (número 0) está guardada;

Esse nó vai ter também várias funções associadas, como ***expandeNode()*** que vai retornar uma lista com todos os filhos do nó. E ***moveBlankPosTo()*** que é a função que executa o movimento, ou seja, troca a peça vazia por uma peça adjacente. Também usámos outra classe ***Search*** que vai guardar todos os algoritmos, a configuração inicial e final, verificar se é possível chegar da configuração inicial a final, e guardar uma fila (*queue*) com um par que vai ser o nó e o custo para atingir um certo nó caso estivermos a usar o algoritmo A*.

A implementação do nó está no ficheiro ***node.py***. O seguinte pedaço de código mostra os valores que cada **nó** vai guardar.

```
class Node:
    def __init__(self, estado, parent):
        self.estado = estado # estado do jogo
        self.parent = parent # pai do nó
        self.moveSet = ''   # lista com os movimentos que levaram o jogo do estado inicial a este atual estado
        self.blankPos = estado.index(0)
```

Implementação da classe Search

```
class Search:
    estadoInicial = None # nó inicial- ainda nao inicializado
    estadoFinal = None   # nó final - ainda nao inicializado
    maxNumberOfNodesStored = 0
    solvability = False
    solution = ''
    visited = dict() # permite adicionar, remover e verificar se contém item em O(1)
                     # o dict terá como chave uma representação em string do nó < str(node) > e valor uma string vazia (não nos interessa o valor)

    def __init__(self, estadoInicial, estadoFinal):
        self.estadoInicial = estadoInicial
        self.estadoFinal = estadoFinal
        self.solvability = self.haSolucao(self.estadoInicial.estado, self.estadoFinal.estado)
```

Implementação da verificação se uma configuração inicial consegue chegar a uma final

Para verificar se uma configuração tem solução nós usamos 3 funções, ***haSolucao(), blankRow()*** e ***inversions()***.
A função ***haSolucao()*** retorna verdadeiro apenas se houver solução para tanto a configuração inicial e final. Para haver solução o numero de inversões em cada configuração tem que ser par e a célula vazia tem que estar numa linha de número ímpar a contar a partir do fim (1ª ou 3ª linha a contar do fim) ou se a célula vazia estiver numa linha de número par (2ª ou 4ª linha a contar do fim) e o número de inversões ímpar.
A condição de solubilidade, foi apresentada nos slides das aulas, ficheiro (solvability.pdf).

```
def haSolucao(self, estadoInicial, estadoFinal):
    blankRowI = self.blankRow(estadoInicial)
    blankRowF = self.blankRow(estadoFinal)
    inversionsI = self.inversions(estadoInicial)
    inversionsF = self.inversions(estadoFinal)
    return (((blankRowI%2==1) == (inversionsI%2 == 0)) == ((blankRowF%2==1) == (inversionsF%2 == 0)))

  def blankRow(self, config):
    return (config.index(0)//4)

  def inversions(self, config):
    totalInversoes = 0
    for i in range(len(config)):
      for j in range(i+1,len(config)):
        if(config[i] > config[j] and config[j] > 0):
          totalInversoes += 1
    return totalInversoes
```

Implementamos também duas funções simples de interação de objetos do tipo ***search, isSolution()*** que verifica se um nó é solução que se pretende e ***getMaxNumberOfNodesStored()*** que retorna o numero máximo de nós guardados numa pesquisa.

As principais funções onde são implementadas as várias estratégias de pesquisa são: **BFS()**, **DFS()**, **IDFS()**, **Greedy()**, **A*()**, onde a Greedy e A*, usam duas heurísticas **getMisplacedTiles()** e **getManhattanDistance()**.

Breve explicação da implementação de cada função:

- ***BFS()*** - Corresponde a pesquisa em banda, onde sao expandidos todos os nodes descendentes, antes de passar para a proxima geracao de descendentes.

Utilizamos uma queque (FIFO), onde colocamos os descendentes, e fazemos a verifcacao de cada um, se nao encontramos solucao expadimos para uma nova geracao(outro nivel da arvore de pesquisa) e voltamos a colocar na queque, e assim sucessivamente, ate encontrar mos a solucao.

- ***DFS()*** - Corresponde a pesquisa em profundidade. Primeiro expandimos o nó, vemos se o primeiro nó é solução, se não for, expandimos a próxima geração vemos se é solução, etc. Neste caso utilizamos uma *stack* (LIFO), e fazemos a verificação de ciclos, com um dicionário auxiliar onde colocamos a representação em string do ***nó***, que visitamos e depois sempre que expandimos um nó verificamos se já está esse nó no dicionário.

- ***IDFS()*** - Usámos a mesma filosofia que no DFS() só que desta vez, usámos um limite, e só deixamos que a pesquisa chegue a determinado nível de profundidade, se não encontramos solução, aumentamos o limite, e repetimos a pesquisa.

- ***Greedy()*** - É a primeira estratégia guiada, a implementação e muito parecida a DFS, mas desta vez como colocamos os nós e o custo, numa *priority queue*, em que o nó de menor custo está no topo (no caso da heurística **peças fora do sítio**, vamos expandir o nó que minimiza a função, que significa ter menos peças fora do lugar).

- ***A*()*** - Idêntico ao *greedy*, só que tomamos em conta não apenas o custo do nó futuro, mas também o custo para atingir o nó atual a partir da raiz.

Estrutura do código

O código está dividido em três ficheiros, ***jogo.py***, ***node.py*** e ***search.py***. Para executar o programa executa-se o ficheiro ***jogo.py*** que recebe a configuração inicial , final e a informação sobre o tipo de pesquisa. A primeira linha de input recebe a configuração inicial do tabuleiro, a segunda linha recebe a configuração final do tabuleiro. Se for possível atingir a solução, o programa avança e pergunta qual é o tipo de pesquisa que o utilizador deseja fazer.

5. Resultados

Estes resultados foram obtidos a partir das configurações dadas como exemplo.

Configuração inicial: 1 2 3 4 5 6 8 12 13 9 0 7 14 11 10 15

Configuração final: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0

Estratégia	Tempo(Segundos)	Espaço	Solução?	Profundidade
DFS	*N/A*	*N/A*	Não	*N/A*
BFS	0.563	8789	Sim	12
IDFS	7.770	12	Sim	12
GREEDY Misp	0.015	255	Sim	42
GREEDY ManDis	0.002	18	Sim	12
A* Misp	0.004	77	Sim	12
A* ManDis	0.003	31	Sim	12

6. Comentários Finais e Conclusões

De acordo com os resultados obtidos podemos ver que, não temos uma estratégia que seja melhor que todas as outras, embora possamos dizer que pelo menos a DFS, não obteve resultado. Dentro das outras estratégias, podemos dizer que em termos de tempo de execução o GREEDY ManDis (com a heurística Distancia de Manhattan), com 0.002 segundos, e com 18 nós apenas utilizados em memória. Se por outro lado valorizar-mos imenso a memória, e não tanto o tempo de execução, IDFS chegou à resposta com apenas 12 nós em memória.

Na nossa opinião, tendo em conta os resultados obtidos, GREEDY ManDis, mostrou ser a melhor estratégia para ser utilizada, porque encontrou o resultado rapidamente com um custo bastante reduzido de memória.

Um ponto a considerar também, é que estes resultados foram apenas obtidos com as configurações inicial e final do exemplo, para uma configuração geral, mais testes teriam de ser feitos, para tentar encontrar a melhor estratégia estatisticamente.

Com isto, percebemos a importância da qualidade da heurística, pois como vimos a heurística peças fora do sítio, no algoritmo *greedy* usou bastante espaço a mais em relação a heurística de distância de Manhattan.

7.Referencias bibliograficas

- Slides do moodle (solvability.pdf)
- Artificial Intelligence A Modern Approach, Stuart Russel,Peter Norvig
- https://python.plainenglish.io/a-algorithm-in-python-79475244b06f
- https://pt.wikipedia.org/wiki/O_jogo_do_15