Relatório
Puzzle Sokoban

Grupo 009
Ana Alferes - nº 50035
Alexandre Nascimento - nº 50002
Rodrigo Nóbrega - nº 50011

1. Formulação de problema

1.1 Representação do estado do jogo

O tabuleiro é representado por uma grelha que com L linhas e C colunas, para além disso tem N caixas e N locais alvo. O número de linhas e colunas irá depender das paredes. Todos os objetos são representados por um tuplo (l, c) com as coordenadas e todos os objetos do mesmo tipo são agrupados dentro de um tuplo. Temos como exemplo o estado inicial:

arrumador = (3, 3)
caixas = ((2, 2),)
paredes = ((0,0), (0,1), (0,2), (0,3), (0,4), (1,0), (2,0), (3,0), (4,0), (4,1), (4,2), (4,3), (4,4), (1,4), (2,4), (3,4))
alvos = ((1, 1),)

1.2 Operadores

Existem 4 movimentos possíveis:

- Arrumador para cima
- Arrumador para baixo
- Arrumador para a esquerda
- Arrumador para a direita

Para o arrumador se mover tem-se que verificar que não existe parede na posição para a qual se quer mover ou se no caso de empurrar uma caixa esta não ir para uma posição livre, sem outra caixa ou parede.
Se arrumador mover para a posição de uma caixa esta também irá se movimentar tendo em conta o sentido do movimento do arrumador e o arrumador irá ocupar a posição da caixa.

2. Heurísticas

1.1 Heurística 1

É visto se alguma das caixas se encontra na posição alvo e se sim subtrai-se 5 caso contrário adiciona 5. No final é devolvido a soma de todos os valores.

1.1 Heurística 2

Inicialmente calculamos a manhattan distance (como vista na fórmula abaixo) entre uma caixa e o alvo mais próximo. Fazemos isto para todas as caixas e somamos todas as distâncias e devolvemos essa soma.

$$d = \sum_{i=1}^{n} |x_i - y_i|$$

1.1 Heurística 3

Verificamos se algumas das caixas se encontra em um canto e devolvemos 100 por cada caixa se isto se verificar pois irá impedir o movimento daquela caixa tornando assim impossível finalizar o puzzle.

1.2 Heurística 4

Consiste na junção das 3 heurísticas anteriores.
Inicialmente calculamos a manhattan distance entre uma caixa e o alvo mais próximo. Fazemos isto para todas as caixas e somamos todas as distâncias ao total.
Seguidamente verificarmos se alguma caixa se encontra em um canto pois isto irá impossibilitar o movimento desta caixa. Se isto se verificar somamos 100 ao total.
Finalmente verificamos se alguma caixa está em uma posição alvo e se sim subtraímos 2 ao total, caso contrário adicionamos 2.

3. Exemplos de execução

Exemplo:

```
prob_sokoban = problem_from_file("puzzle1.txt")
print(prob_sokoban.initial)
print("****")

tent1 = greedy_best_first_graph_search(prob_sokoban, prob_sokoban.h4)
print(tent1.solution())
print(len(tent1.solution()))
print_path(tent1)
```

Neste exemplo será lido o ficheiro “puzzle1.txt” com a função problem_from_file() que irá devolver o problema criado tendo em conta os dados do ficheiro. De seguida é mostrado o estado inicial do puzzle.
Irá se iniciar a pesquisa greedy_best_first_graph_search com o problema criado que foi lido do ficheiro e a heurística 4. Quando finalizada é mostrado os movimentos necessárias, o número de movimentos e é mostrada a resolução do puzzle passo a passo.

```
fc50035@linux:~/SI/projeto SI$ python3 sokoban.py
#####
#o..#
#.*.#
#..A#
#####
```

↳ É iniciado o programa no terminal com o puzzle que queremos ver resolvido indicado no ficheiro e é nos mostrado o início do puzzle.

```
****
['mover para a esquerda', 'mover para cima', 'mover para a direita', 'mover para cima', 'mover para a esquerda']
5
```

↳ Começamos por indicar os movimentos realizados pelo arrumador e em seguida o nº de jogadas que têm de ser realizadas pelo arrumador

```
#####
#o..#
#.*.#
#..A#
#####

#####
#o..#
#.*.#
#.A.#
#####

#####
#o*.#
#.A.#
#...#
#####

#####
#o*.#
#..A#
#...#
#####

#####
#o*A#
#...#
#...#
#####

#####
#@A.#
#...#
#...#
#####
```

→ Em seguida é mostrado a resolução do puzzle pela representação indicada no início

No final o terminal terá este aspecto:

```
fc50035@linux:~/SI/projeto SI$ python3 sokoban.py
#####
#o..#
#.*.#
#..A#
#####

****
['mover para a esquerda', 'mover para cima', 'mover para a direita', 'mover para cima', 'mover para a esquerda']
5
#####
#o..#
#.*.#
#..A#
#####

#####
#o..#
#.*.#
#.A.#
#####

#####
#o*.#
#.A.#
#...#
#####

#####
#o*.#
#..A#
#...#
#####

#####
#o*A#
#...#
#...#
#####

#####
#@A.#
#...#
#...#
#####
```

4. Análise dos Algoritmos

H1:

Heurística simples, bom para resolver puzzles mais simples pois nestes garante a solução mais eficiente (menos movimentos), no entanto com o aumento da complexidade dos puzzles prova-se ineficiente como se pode verificar no exemplo abaixo pois não garante o menor número de movimentos usando o algoritmo astar_search.

H2:

Oferece melhor desempenho que a heurística H1 em todos os puzzles, mas que peca comparativamente com as H3 e H4, principalmente com o aumento da complexidade dos puzzles

H3:

Oferece um grande aumento de eficiência comparativamente com as heurísticas anteriores. Pode-se concluir que este aumento se deve ao facto de conseguir excluir muitos estados que iram tornar a resolução do puzzle impossível.

H4:

Heurística desenvolvida, sendo a com melhor desempenho pois é a que necessita de menos tempo e cria menos estados, comparativamente com as outras heurísticas, dando com o algoritmo de procura astar_search o número mínimo de movimentos necessários

Ex:**“puzzle5.txt”**

Número mínimo de movimentos - 72

Heurística	**Função de procura**	**Número de movimentos**	**Número de estados espandidos**
H1	greedy_best_first_graph_search	76	32945
H1	astar_search	76	27894
H2	greedy_best_first_graph_search	76	8251
H2	astar_search	72	31931
H3	greedy_best_first_graph_search	80	7121
H3	astar_search	72	5148
H4	greedy_best_first_graph_search	76	3122
H4	astar_search	72	4334

5. Conclusão

Pode-se concluir que a heurísticas H3 e H4 são mais eficiente que as H1 e H2, pois para além de demorarem menos tempo a executar, têm um menor número de estados expandidos e devolvem o menor número de movimentos possíveis. Mesmo assim provam-se incapazes de resolver de modo eficaz certos puzzles tal como o “puzzle3.txt” dado pelo professor.

Podemos também concluir em relação às funções de procura que a função greedy_best_first_graph_search comparativamente à depth_first_graph_search oferece muito melhor desempenho pois apesar de não garantir a melhor solução, consegue na maioria das vezes oferecer uma solução melhor muito mais eficácia. A função astar_search também irá oferecer melhor desempenho ao comparar com a breadth_first_search pois oferece a melhor solução não necessitando de percorrer todos os movimentos possíveis. A eficacia tanto da função greedy_best_first_graph_search como da astar_search depende muito da heurística usada.

Uma grande dificuldade com este puzzle é o aumento exponencial do número de possíveis movimentos o que lhe oferece uma grande complexidade.