INTELIGÊNCIA ARTIFICIAL

Match The Tiles

Grupo 39

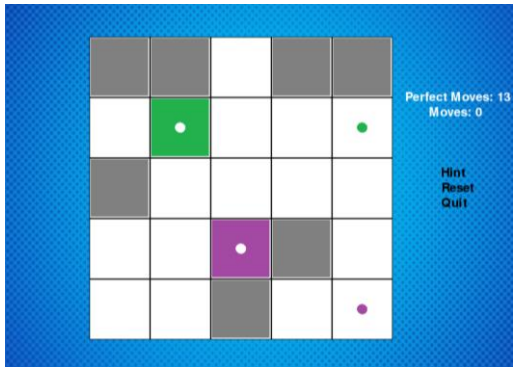
Mariana Ramos – up201806869
Pedro Ferreira – up201806506
Pedro Ponte – up201809694

Especificação do Projeto

O jogo consiste na existência de um tabuleiro contendo diferentes tipos de “tiles”, correspondendo cada tile a uma célula do tabuleiro.

Existem três tipos de tiles no jogo, onde cada uma apresenta uma cor diferente que os relaciona com o seu objetivo no jogo. Tiles preenchidas com uma cor escura correspondem a paredes, tiles com um ponto no centro correspondem a tiles objetivo e tiles pintadas com um círculo no centro correspondem a tiles jogáveis.

O objetivo do jogo é colocar as tiles jogáveis nas posições onde se encontram as tiles objetivo com a cor correspondente. Para tal, o jogador pode deslocar as peças para cima, baixo, esquerda e direita. Os movimentos são sincronizados, portanto, deve-se usar os tiles fixos existentes para criar espaços entre os tiles e resolver o puzzle.



Formulação do Problema

Estado de Representação

O tabuleiro de jogo é representado por uma matriz B quadrangular com Y colunas e Y linhas, 4 <= Y <= 6. Cada célula pode ter os seguintes valores:

- ‘X’, no caso de ser uma parede;
- ‘IB’, ‘IG’, ‘IO’, ‘IP’, ‘IR’ ou ‘IY’ no caso de ser uma célula jogável;
- ‘FB’, ‘FG’, ‘FO’, ‘FP’, ‘FR’ ou ‘FY’ no caso de ser uma célula destino;
- ‘-’ no caso de ser uma célula vazia.

- **Estado Inicial**

  Matriz B com estado inicial desejado. Por exemplo:

```
B = [  [ 'X' , ‘X’ , '-’ , ‘X’ ],
       [ ‘-’ , ’X’ , ’-’ , ’IP’ ],
       [ ‘FP’ , ‘X’ , ‘IP’ , ‘X’ ],
       [ ‘-’ , ’-’ , ’-’ , ’FP’ ] ]
```

- **Estado Objetivo**

  Matriz F sem estados finais (FZ) sozinhos. Por exemplo:

```
F = [  [ 'X' , ‘X’ , '-’ , ‘X’ ],
       [ ‘-’ , ’X’ , ’-’ , ’-’ ],
       [ ‘IPFP’ , ‘X’ , ‘-’ , ‘X’ ],
       [ ‘-’ , ’-’ , ’-’ , ’IPFP’ ] ]
```

Formulação do Problema

Operadores

Legenda:

IZ - peça que se move

IY - outra peça

X - parede

FK - célula final (Se IZ estivesse em cima de uma antes do movimento), K ∈ {Z, Y}

Nome	Pré-condição	Efeitos	Custo
Up	∃IZ, B[IZx , IZy-1] ∉ { “X” , ”IY” } ∧ IZy > 0	∀IZ, While (B[IZx-1, IZy] ∉ { “X” , ”IY” }) do B[IZx , IZy-1]=IZ; B[IZx, IZy] = “-” or B[IZx, IZy] = FK	1
Down	∃IZ, B[IZx , IZy + 1] ∉ { “X” , ”IY” } ∧ IZy < B.size	∀IZ, While (B[IZx+1,IZy] ∉ { “X” ,”IY”}) do B[IZx , IZy+1] = IZ; B[IZx, IZy] = “-” or B[IZx, IZy] = FK	1
Left	∃IZ, B[IZx-1 , IZy] ∉ { “X” , ”IY” } ∧ IZx > 0	∀IZ, While (B[IZx , IZy-1] ∉ { “X” , ”IY” }) do B[IZx-1 , IZy] = IZ; B[IZx, IZy] = “-” or B[IZx, IZy] = FK	1
Right	∃IZ, B[IZx+1 , IZy] ∉ { “X” , ”IY” } ∧ IZx < B.size	∀IZ, While (B[IZx , IZy+1] ∉ { “X” ,”IY”}) do B[IZx+1 , IZy] = IZ; B[IZx, IZy] = “-” or B[IZx, IZy] = FK	1

Formulação do problema

Heurísticas (ficheiro heuristics.py)

Para os algoritmos de pesquisa gananciosa e A*, criamos 3 heurísticas diferentes para podermos testar os algoritmos e chegar a algumas conclusões relativamente à eficiência de cada uma delas.

- *heuristic1*: heurística em que se atribuí um menor valor aos nós cujos tabuleiros contêm mais peças na mesma linha ou coluna que a respetiva peça final;
- *heuristic2*: heurística em que se atribuí um menor valor aos nós cujos tabuleiros contêm mais peças na respetiva posição final;
- *heuristic3*: heurística em que se atribuí um menor valor aos nós em que os tabuleiros têm menor distância de Manhattan entre as posições atuais e finais de cada peça que ainda não se encontrem na respetiva posição final.

Formulação do problema

Heurísticas (Continuação)

Usadas na pesquisa gananciosa:

- Depois de analisados os resultados da conjugação das várias heurísticas, concluímos que a melhor heurística a utilizar neste algoritmo seria a *heuristic1*.

Usadas no algoritmo A*:

- Através da análise dos resultados da conjugação das várias heurísticas, concluímos que os resultados mais eficientes são obtidos quando conjugamos a *heuristic1* com a distância desde o nó inicial até ao nó atual (*depth*).

Os resultados obtidos para as diversas conjugações de heurísticas e que serviram de base para a nossa escolha podem ser vistas nos anexos no final desta apresentação (ver anexo I para a pesquisa gananciosa e anexo II para o algoritmo A*).

Detalhes da implementação

Linguagem de programação

Python, recorrendo ao pacote *pygame* para representação da interface do jogo.

Ambiente de desenvolvimento

VSCode/ Spyder

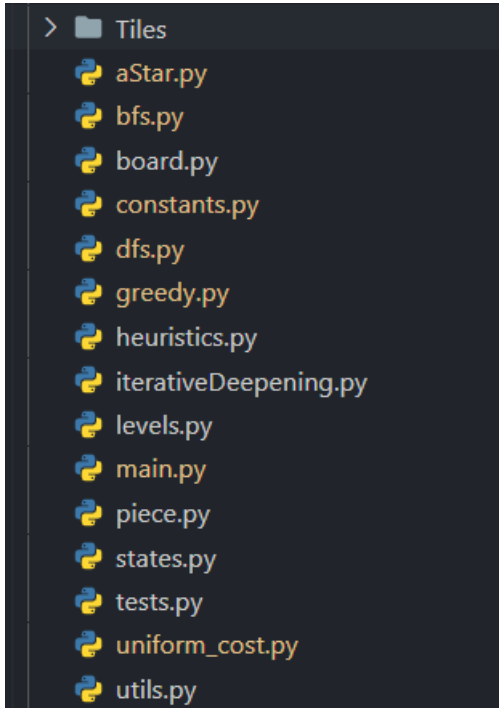
Estruturas de dados

Board, que contém o estado atual do tabuleiro;

Node, para guardas as informações dos diferentes nós gerados durante a pesquisa;

Piece, que guarda as informações sobre a localização das peças e a respetiva cor.

Estrutura de ficheiros



Algoritmos de pesquisa implementados

- **Pesquisa primeiro em profundidade (DFS):** A DFS consiste na expansão dos nós da árvore, começando na sua raiz, e aprofundando progressivamente a cada iteração, até chegar ao nó de profundidade máxima, retrocedendo depois para visitar os restantes.
  - ✓ De modo a tornar o algoritmo um pouco mais eficiente removemos todos os ciclos evitando repetir estados de tabuleiro já visitados. No nosso problema, um nó é explorado em profundidade até se encontrar uma solução, terminando a pesquisa. Desta forma é sempre encontrada uma solução, mas não é garantida a solução ótima (menor número de movimentos).
- **Pesquisa primeiro em largura (BFS):** A BFS, por sua vez, consiste na expansão dos nós em largura, ou seja, nível a nível, apenas passando para uma profundidade d+1 quando todos os nós de d tiverem sido visitados.
  - ✓ No caso do presente problema, os nós são explorados até ser encontrada a solução ótima.
- **Iterative Deepening:** Com o algoritmo iterative deepening, efetuamos uma DFS com um nível de profundidade limitado e fomos aumentando esse nível de profundidade até encontrar uma solução.
  - ✓ Este algoritmo a partir do nível 15 (nível com número ótimo de movimentos 11) começou a tornar-se demasiado demorado apesar de encontrar a solução ótima.

Algoritmos de pesquisa implementados

- **Pesquisa de custo uniforme:** Esta pesquisa é semelhante à BFS, no sentido em que explorará os nós por níveis de profundidade, mas, em vez de os expandir por ordem, estes irão ser ordenados segundo uma função de custo, sendo explorados os nós de um determinado nível por ordem desse custo (do menor para o maior).

  ✓ Como o custo de qualquer movimento neste caso é sempre 1, então este algoritmo torna-se igual à pesquisa em largura.

- **Pesquisa gananciosa:** O algoritmo ganancioso explora os nós em profundidade, utilizando uma heurística gananciosa para escolher o próximo nó a explorar.

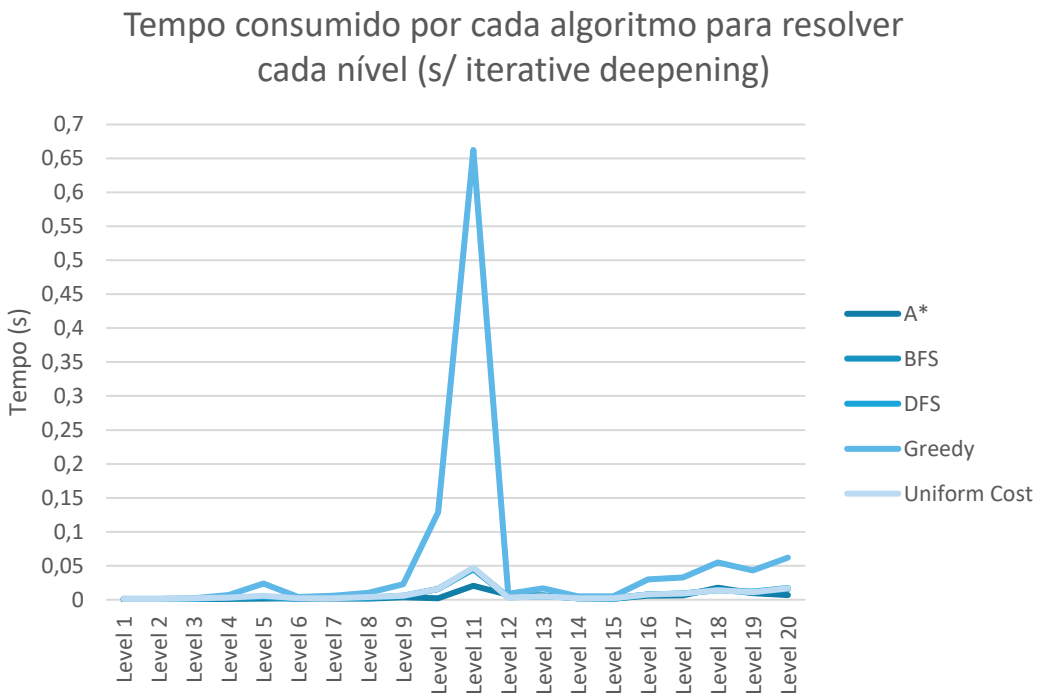
- **A*:** Este algoritmo combina o custo para chegar até ao nó atual (f(n)), com a estimativa do custo de avançar para o próximo nó (g(n)), sendo f(n)= g(n)+h(n), seguindo sempre o caminho que minimiza f(n). O algoritmo A* necessita de manter todos os nós em memória enquanto procura a solução uma vez que efetua uma pesquisa exaustiva. Assim enquanto que a complexidade temporal é igual à do greedy, o algoritmo A* têm uma complexidade espacial maior O(b^m).

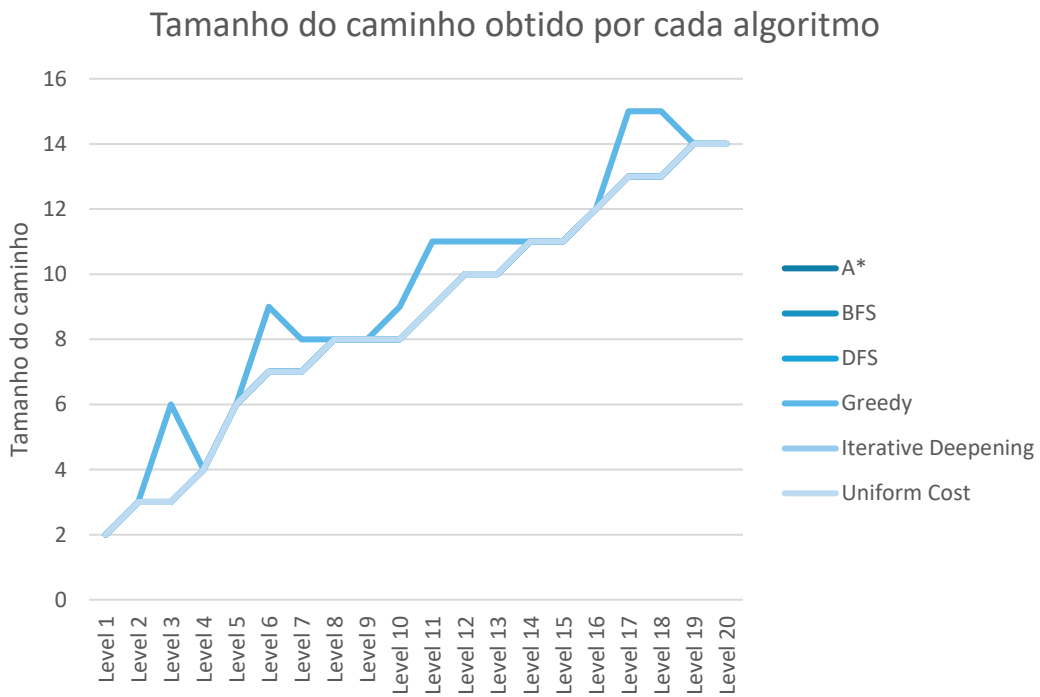
Enquanto que o algoritmo A* alcança sempre a solução ótima e é completo, o algoritmo greedy não é completo, o que significa que pode optar por caminhos que não alcancem o estado objetivo. (Ver anexo I e II para comparação de resultados).

Devido às limitações do hardware, implementamos um *max-depth* nos algoritmos.

Resultados Experimentais

Mediram-se os tempos de execução para os 20 primeiros níveis do Match The Tiles, com os 6 algoritmos implementados (ver anexo III para mais resultados).





Conclusão

O projeto desenvolvido foi relevante para a consolidação dos conteúdos lecionados na UC de Inteligência Artificial.

Concluímos, tal como seria esperado, que os algoritmos de pesquisa informada (Greedy e A*) são mais eficientes do que os métodos de pesquisa não informada (BFS, DFS, *Uniform Cost*, *Iterative Deepening*). Dentro dos algoritmos de pesquisa informada, concluímos que para os puzzles construídos, o A* se demonstra como mais eficiente. Relativamente ao *Iterative Deepening*, este demonstrou ser o menos eficaz, uma vez que necessita de expandir muitos mais nós do que os restantes para conseguir atingir os mesmos resultados.

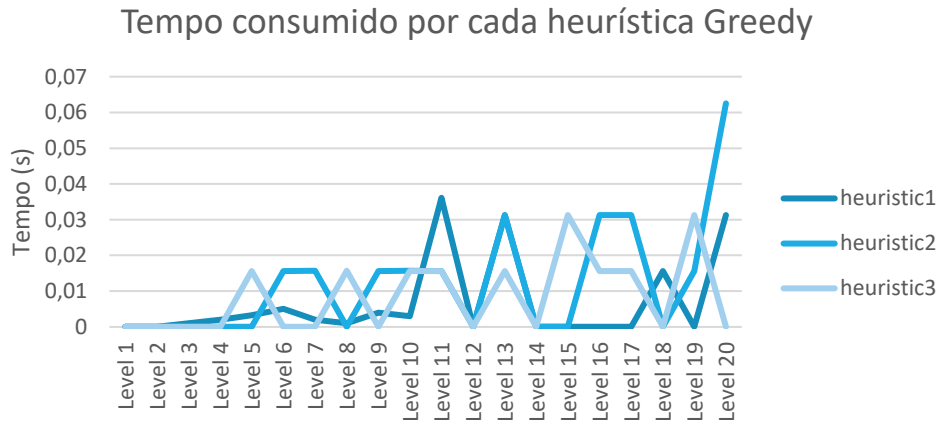
Entendemos também a importância de escolher heurísticas adequadas, sendo que esta foi uma dificuldade com que nos deparamos no desenvolvimento do projeto. Desenvolvemos 3 diferentes heurísticas para podermos comparar os respetivos resultados quando aplicadas aos algoritmos Greedy e A*, tendo concluído que a *heuristic1* é a mais eficaz para ambos os algoritmos. Por outro lado, chegamos também à conclusão que a *heuristic3* não é boa, uma vez que quando aplicada ao A*, para alguns níveis, faz com que não se atinja uma solução ótima.

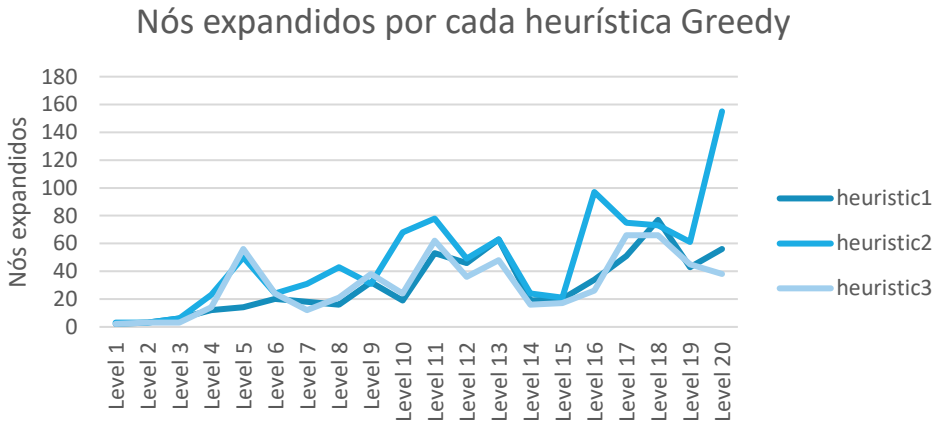
Referências

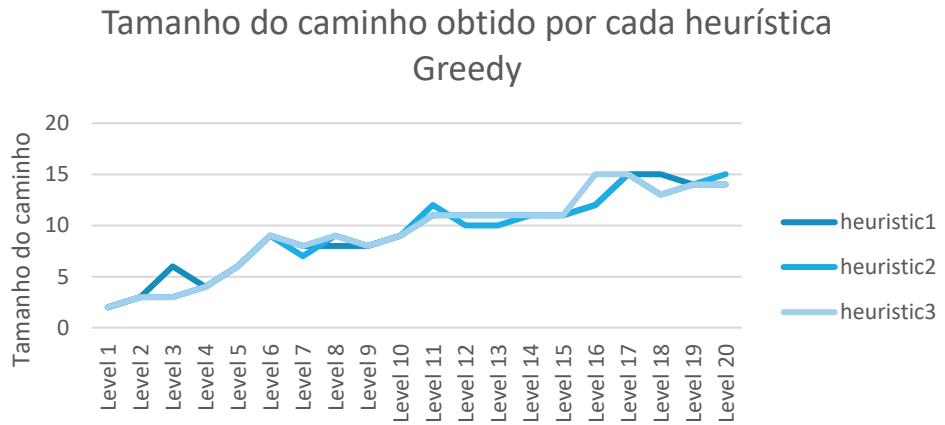
- Link para a página do jogo na Google Play;
- Exemplo para desenvolvimento gráfico;
- Slides das aulas teóricas e teórico-práticas para desenvolvimento dos diferentes algoritmos.

Anexos

Anexo I



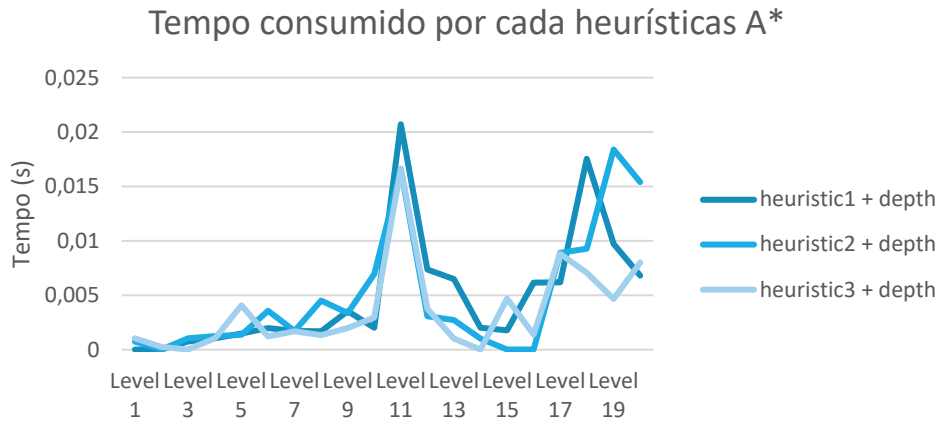


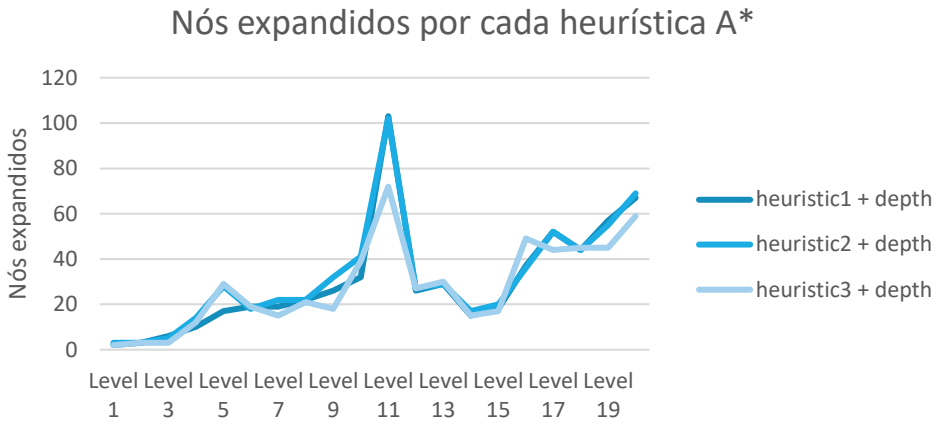


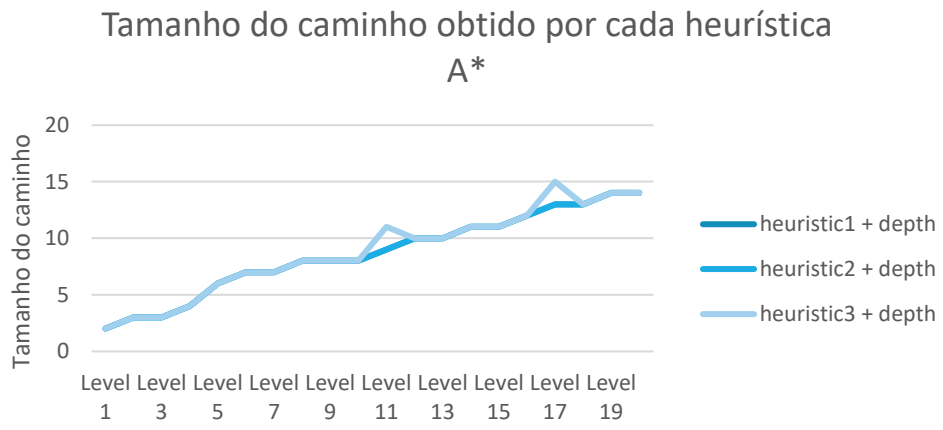
Verifica-se que a *heuristic2*, apesar de os caminhos por ela obtidos serem mais próximos dos caminhos ideais, demora na maioria das vezes mais tempo do que a *heuristic1* e necessita de expandir mais nós do que as restantes heurísticas.

Os resultados obtidos para as *heuristics* 1 e 3 são muito semelhantes em termos de tempo consumido, mas verifica-se que a *heuristic*3 em certos níveis (como o nível 5) necessita de expandir bastantes mais nós para atingir o mesmo resultado. Para além disso, esta heurística, como veremos no anexo seguinte, não é totalmente fiável.

Anexo II







Verifica-se que a *heuristic3* para os níveis 11 e 17 não apresenta como solução o caminho ideal, pelo que não é uma boa heurística.

Comparando as *heuristics* 1 e 2, verifica-se que, apesar de apresentarem alguns resultados bastante semelhantes, a *heuristic*1 tende a demorar menos tempo para resolver os níveis do que a *heuristic*2.

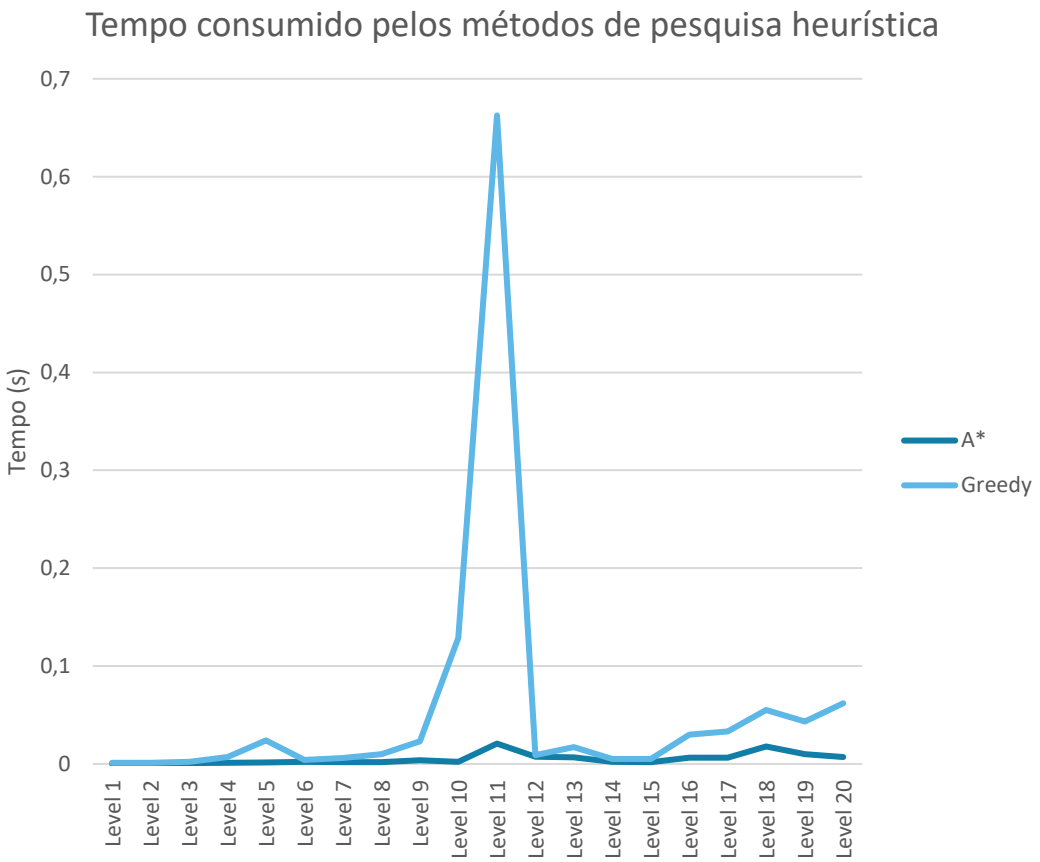
Relativamente aos nós expandidos, ambas as heurísticas apresentam resultados muito parecidos para todos os níveis.

Anexo III


Tempo consumido por cada algoritmo para resolver cada nível
Tempo (s)
120
100
80
60
40
20
0
Level 1
Level 2
Level 3
Level 4
Level 5
Level 6
Level 7
Level 8
Level 9
Level 10
Level 11
Level 12
Level 13
Level 14
Level 15
Level 16
Level 17
Level 18
Level 19
Level 20
A*
BFS
DFS
Greedy
Iterative Deepening


Tempo consumido por cada algoritmo para resolver cada nível (s/ iterative deepening)
Tempo (s)
0,7
0,65
0,6
0,55
0,5
0,45
0,4
0,35
0,3
0,25
0,2
0,15
0,1
0,05
0
Level 1
Level 2
Level 3
Level 4
Level 5
Level 6
Level 7
Level 8
Level 9
Level 10
Level 11
Level 12
Level 13
Level 14
Level 15
Level 16
Level 17
Level 18
Level 19
Level 20
A*
BFS
DFS
Greedy
Uniform Cost

Anexo III (continuação)



Anexo III (continuação)

Anexo III (continuação)