**Métodos de Pesquisa Heurística para Resolução de Problemas**

**Jogo: “Folding Blocks”**

**Trabalho realizado por:**

Bernardo Moreira – up201604014

Filipe Nogueira – up201604129

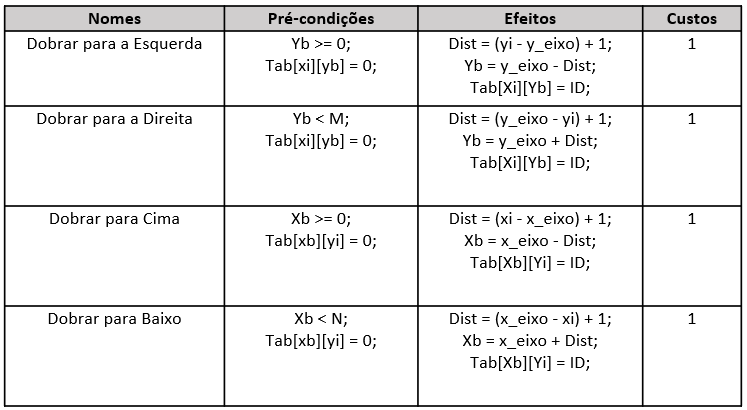
Francisco Pereira – up201605306

**Introdução**

* Neste projeto é pretendido implementar diversos algoritmos de pesquisa capazes de vencer uma série de níveis do jogo “**Folding Blocks”**.
* Este jogo, inicialmente composto por um tabuleiro com uma ou mais peças, tem como objetivo, ocupar todos os espaços dos tabuleiros utilizando essas peças. Esta utilização é feita através da seleção da peça a jogar, seguida de uma transformação simétrica dessa mesma peça de acordo com o operador selecionado, isto é, cada movimento vai **duplicar** o tamanho da peça na direção escolhida.

**Formulação do Problema**

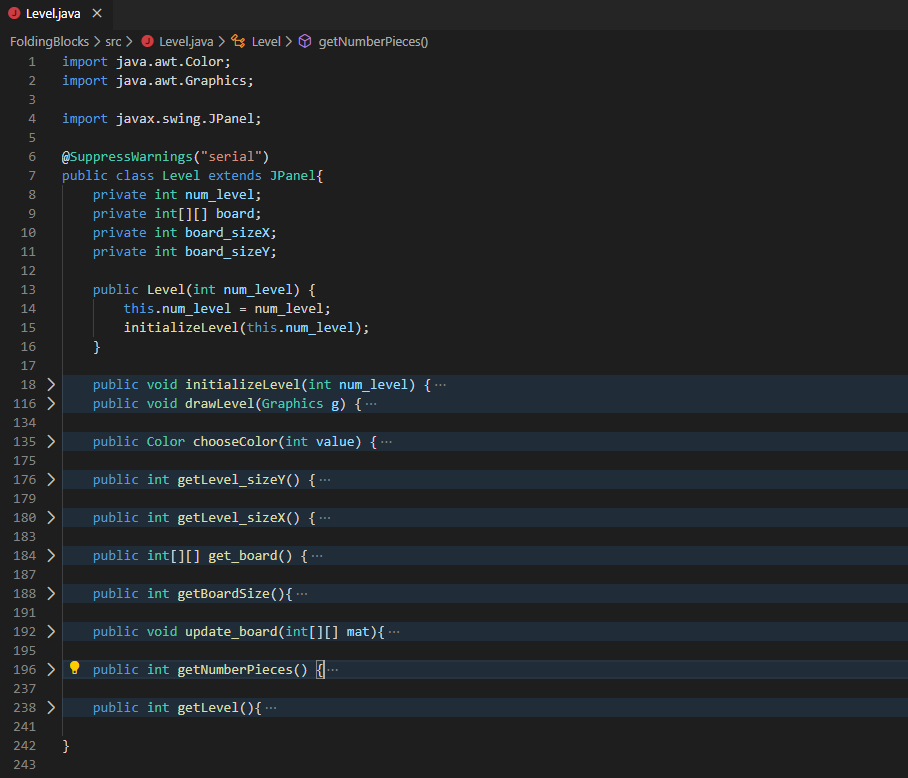
* **Representação do estado:** O tabuleiro do jogo é representado por uma matriz com tamanho variável (de acordo com o nível). Esta matriz é do tipo int[][], cujo valores podem ser:
  + 0 - caso de espaços vazios;
  + N - sendo 0 < N < Tamanho máximo do tabuleiro, onde a cada N corresponde um tipo de peça.
* **Teste objetivo:** O jogo acaba quando o tabuleiro não tiver espaços vazios. Ou seja, quando em todas as posições do tabuleiro x e y, board[x][y] seja diferente de 0.
* **Operadores:**



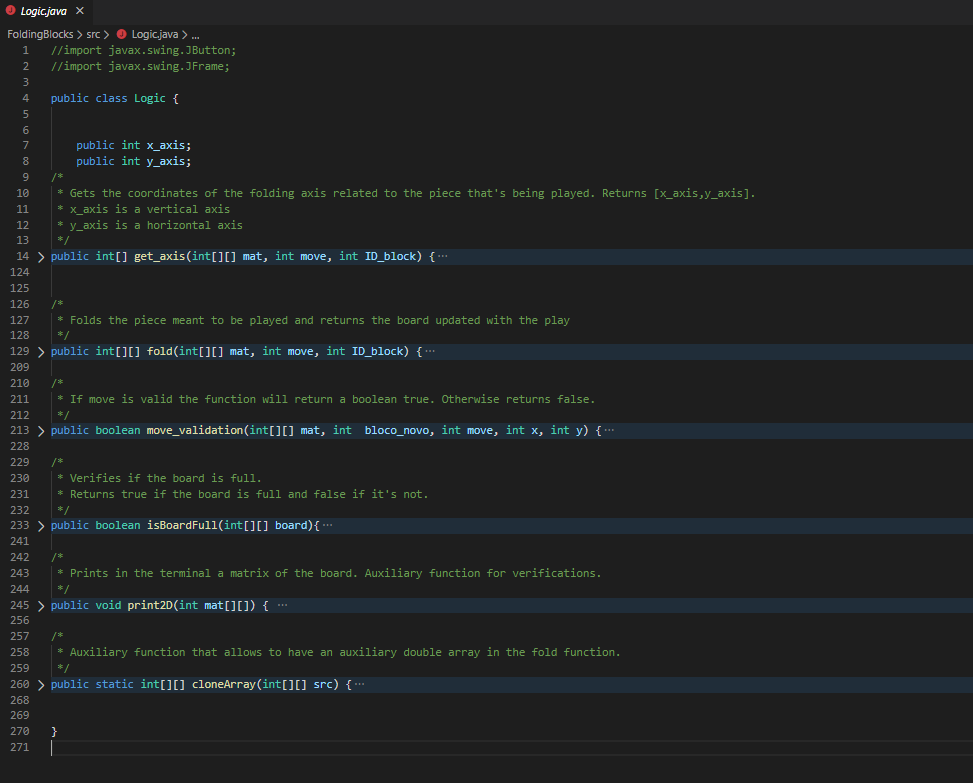
* ID - referente à peça a ser jogada;
* Tab[ ][ ] - valor na posição de cada tabuleiro (pode ter os seguintes valores: 0 - se estiver livre / ID - numero referente à peça a ser jogada);
* Xi / Yi - coordenada referente à linha/coluna (respetivamente) do bloco atual;
* Xb / Yb - coordenada referente à linha/coluna (respetivamente) do bloco novo;
* x\_eixo / y\_eixo - coordenada referente à linha/coluna (respetivamente) do eixo de simetria;
* Dist - distância ao eixo de simetria;
* N x M - dimensões do tabuleiro de jogo (linhas/colunas);

**Implementação do Jogo**

* Linguagem escolhida para desenvolvimento do código: **Java**
* Trabalho realizado em **Eclipse** & **VSCode**
* Código foi dividido pelas seguintes classes:
  + Main.java
  + Game.java
  + Level.java
  + Logic.java
  + Algoritmo.java
* A classe Level trata de definir, e desenhar os níveis. Trata da atribuição de cores consoante os valores da matriz e trata de atualizar a mesma.

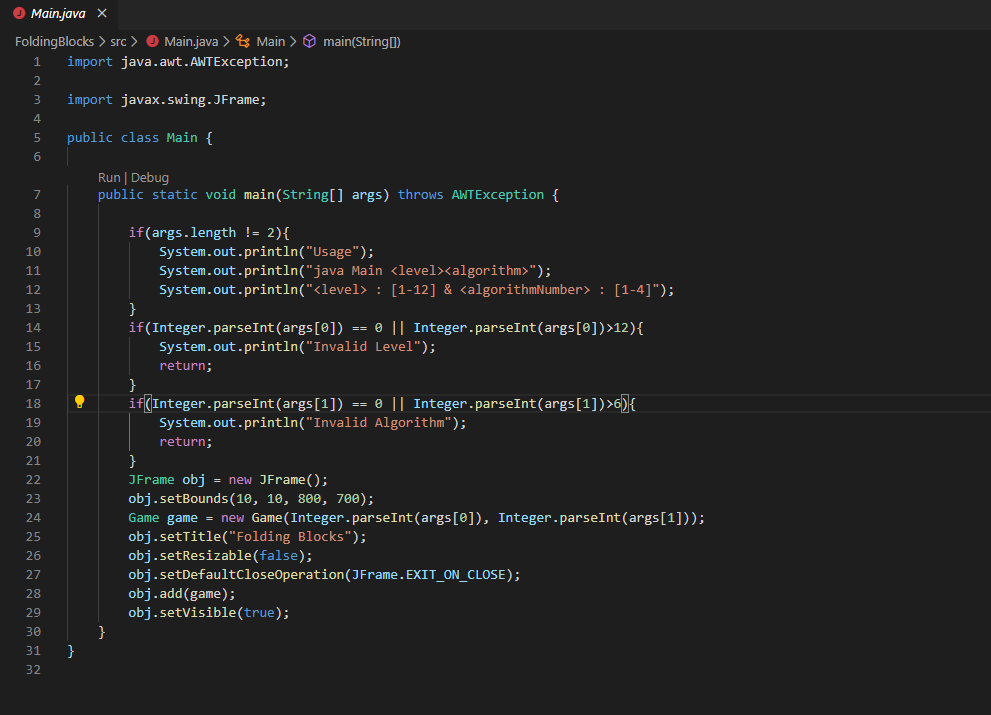


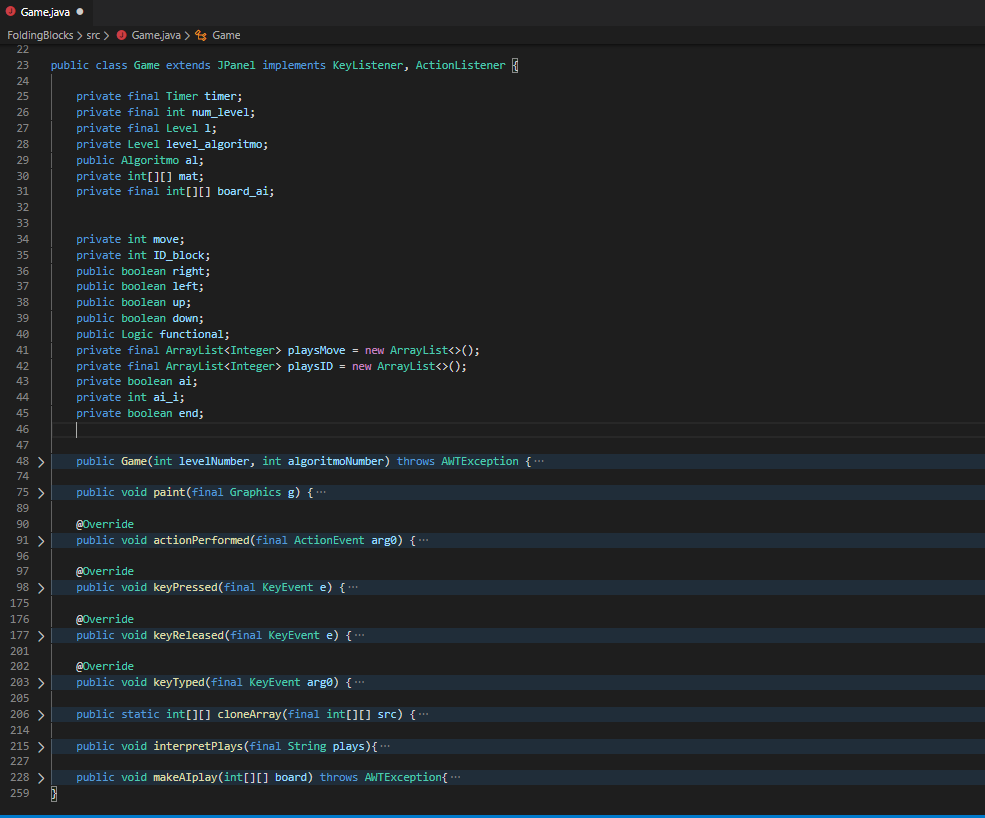
* A classe Logic verifica a lógica de jogo. Trata de fazer as jogadas, bem como as verificações de jogada e de fim de jogo.



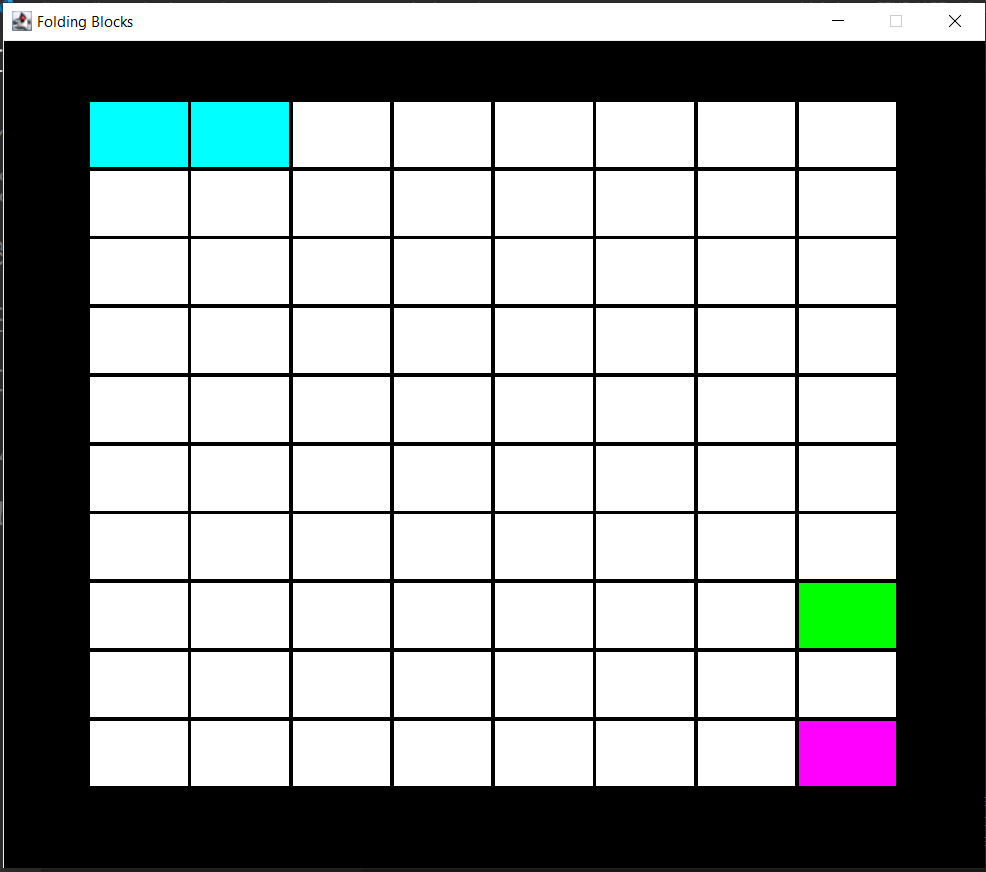
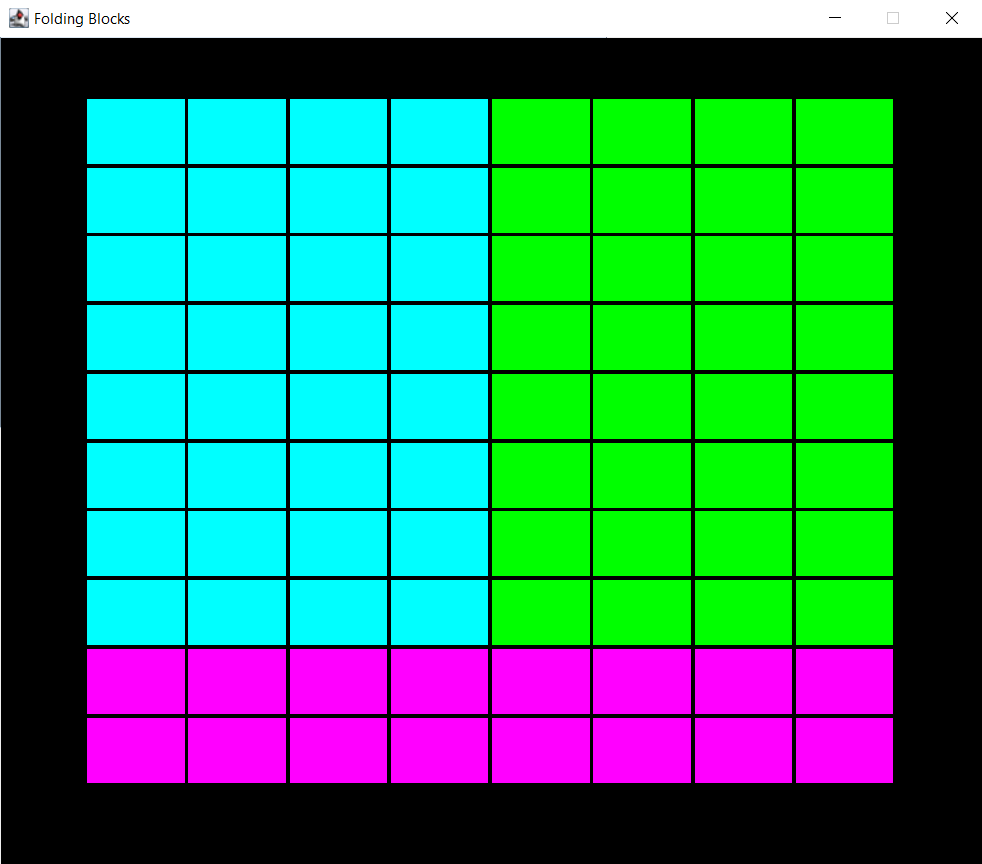
* Finalmente, tanto a classe Main como a Game servem para funcionalidades da interface. Permite a criação de uma janela para o jogo, bem como permite à classe Level que desenhe nesta mesma interface. Inicialmente implementamos também deteção de pressão de teclas, para testarmos as funcionalidades do jogo e permitir ao utilizador jogar o mesmo.

A classe Game dá também início ao algoritmo de pesquisa utilizado (conforme a escolha do utilizador) para a descoberta de solução, assim como interpreta as jogadas e as realiza autonomamente.





O objetivo de jogo, como já mencionado é completar todo o tabuleiro com as peças existentes no mesmo, e é possível através da nossa interface gráfica facilmente identificar o estado inicial e final do jogo:

**Algoritmos e Heurísticas**

Para a resolução do nosso projeto, de forma a testar a variação de tempo gasto na resolução do problema, número de nós explorados assim como número de jogadas feitas foram avaliadas 5 implementações:

* **“Depth First Search”:** É um algoritmo que tem como estratégia expandir sempre primeiro o nó de maior profundidade. Acaba por se apresentar como uma boa solução para o nosso problema em concreto dada a existência de várias soluções e a irrelevância da sequência em que as peças são jogadas. Aspetos negativos da implementação deste algoritmo no jogo “Folding Blocks” acontece quando este se depara com níveis mais complexos em que existe um maior número de jogadas que podem levar a resoluções impossíveis. Nestes casos o método de pesquisa prova-se ineficiente pois explora demasiados nós irrelevantes devido a falta de critério na seleção destes mesmos.
* **“Greedy”:** Este algoritmo tem como estratégia expandir o nó que está mais perto da solução. Utiliza uma função heurística que devolve um custo estimado do caminho mais curto do estado n para o objetivo (de acordo com a função heurística que lhe é atribuída). É de notar o facto de que esta estratégia não devolve necessariamente a solução ótima.

Com este algoritmo foram utilizadas duas heurísticas diferentes:

* + Heurística2:

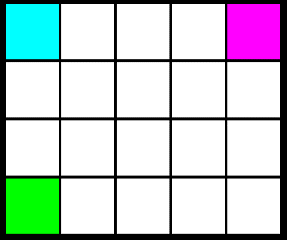
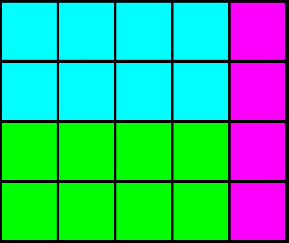
Esta heurística avalia o número de espaços preenchidos e verifica a chegada à solução assim que o valor de h(n) igualar o número de espaços existentes no tabuleiro. Utilizado com um método de pesquisa “Greedy” procura constantemente um movimento que proporcione o maior número de peças jogadas.

* + Heurística3:

O valor retornado por esta função heurística estima, de acordo com o estado do tabuleiro, o número de jogadas até à solução final partindo do pressuposto que todas as peças terão o mesmo tamanho final. É aplicado ao algoritmo de pesquisa “Greedy” tentando sempre minimizar o número de jogadas até ao final da solução.

Esta função sobrestima a solução final, e como tal, não deve ser usada com o algoritmo A\*.

Para o caso do nível 3, o estado inicial e final são os seguintes:

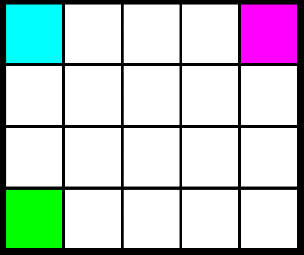
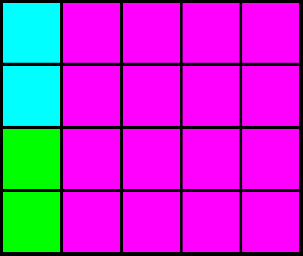
 

* **“A\*”:** O algoritmo A\* combina a pesquisa gulosa com a de custo uniforme minimizando a soma do caminho já efetuado com o mínimo previsto/estimado que falta até à solução. A solução do algoritmo A\* é ótima é completa sendo importante o uso de uma heurística adequada.
  + Heurística4:

A função heurística em cima representada apresenta um valor subestimado do número de jogadas até encontrar solução final. Esta considera o melhor caso de jogadas para um tabuleiro com uma dada dimensão, que seria este ter apenas uma peça, e esta ser expandida ao máximo. Esta heurística apresenta-se admissível para ser usada com o algoritmo A\* dado o facto de nunca sobrestimar a solução final, garantindo juntamente com o custo uma solução ótima ao problema.

É de notar que dadas as características do problema representa uma solução lenta e extensa, explorando por norma muitos nós.

Para o caso do nível 3, o estado inicial e final são os seguintes:

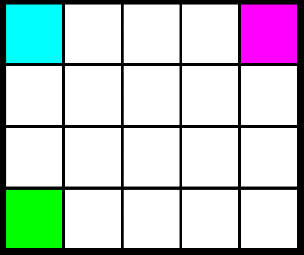
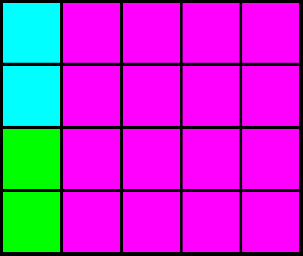
 

É possível notar que o algoritmo adota uma abordagem diferente daquela utilizada no algoritmo “Greedy”. Este opta por expandir ao máximo a peça cor de rosa, devido ao tamanho final maior que esta consegue obter, resultando numa solução com menos jogadas (solução ótima).

* **Implementação própria:** Neste caso o algoritmo atribui a cada nó o valor de acordo com a heurística em baixo representada e dá prioridade aquele com maior valor.
  + Heurística5:

Esta heurística tem por objetivo dar prioridade às peças que conseguem atingir uma maior peça total final (tendo na mesma em conta o novo número de peças que se obtém com cada jogada e o custo). Isto acontece, pois, a solução ótima obtém-se expandindo essa mesma peça ao máximo, dentro dos possíveis, e garantindo a existência de solução, pois vai ser possível cobrir mais espaços livres em menos jogadas.

Tal como o A\*, este algoritmo encontra a solução ótima:

**Comparação de Resultados**

* Tempo:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| TIME (ms) | | | | | |
|  |  |  |  |  |  |
|  | DFS | Greedy H2 | Greedy H3 | A\* | Our Own |
| Level 1 | 4 | 5 | 5 | 5 | 5 |
| Level 2 | 4 | 5 | 5 | 9 | 5 |
| Level 3 | 5 | 5 | 7 | 14 | 5 |
| Level 4 | 5 | 5 | 6 | 7 | 5 |
| Level 5 | 5 | 6 | 7 | 113 | 9 |
| Level 6 | 5 | 7 | 29 | 95 | 7 |
| Level 7 | 5 | 7 | 11 | 296 | 8 |
| Level 8 | 7 | 10 | 9 | 971 | 8 |
| Level 9 | 6 | 8 | 16 | ND | 715 |
| Level 10 | ND | ND | ND | ND | 13284 |
| Level 11 | 11 | 15 | 11366 | ND | 288 |

Através da comparação dos tempos podemos ver que a pesquisa em profundidade (DFS) e o algoritmo Greedy vão por norma apresentar maior rapidez na resolução dos problemas. Contrariamente o A\* não se revela tão vantajoso nesse aspeto.

* Nós utilizados:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Number Of Nodes | | | | | |
|  |  |  |  |  |  |
|  | DFS | Greedy H2 | Greedy H3 | A\* | Our Own |
| Level 1 | 4 | 4 | 4 | 11 | 4 |
| Level 2 | 6 | 6 | 9 | 60 | 6 |
| Level 3 | 8 | 8 | 11 | 140 | 7 |
| Level 4 | 5 | 5 | 7 | 34 | 5 |
| Level 5 | 12 | 12 | 26 | 2374 | 26 |
| Level 6 | 13 | 13 | 444 | 1931 | 13 |
| Level 7 | 13 | 13 | 48 | 6651 | 27 |
| Level 8 | 58 | 70 | 36 | 34871 | 25 |
| Level 9 | 17 | 17 | 55 | ND | 22235 |
| Level 10 | ND | ND | ND | ND | 191171 |
| Level 11 | 97 | 85 | 217690 | ND | 5907 |

Por norma o valor de nós utilizados vai estar diretamente relacionado com tempo, dado que quantos mais nós o algoritmo explora mais tempo gastará a encontrar a solução. Podemos reparar que em geral a pesquisa para os algoritmos Greedy e DFS são menos extensas e como consequência mais rápidas, como já foi mencionado acima.

* Jogadas:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Number Of plays | | | | | |
|  |  |  |  |  |  |
|  | DFS | Greedy H2 | Greedy H3 | A\* | Our Own |
| Level 1 | 4 | 4 | 4 | 4 | 4 |
| Level 2 | 6 | 6 | 6 | 6 | 6 |
| Level 3 | 8 | 8 | 8 | 6 | 6 |
| Level 4 | 5 | 5 | 5 | 5 | 5 |
| Level 5 | 12 | 12 | 12 | 12 | 12 |
| Level 6 | 13 | 13 | 13 | 11 | 11 |
| Level 7 | 13 | 13 | 13 | 13 | 13 |
| Level 8 | 10 | 10 | 10 | 10 | 10 |
| Level 9 | 17 | 17 | 17 | ND | 17 |
| Level 10 | ND | ND | ND | ND | 13 |
| Level 11 | 18 | 18 | 18 | ND | 16 |

O número de jogadas utilizadas vai depender do nível em questão e do algoritmo utilizado. O A\* vai oferecer sempre a melhor jogada perante o nível que lhe é apresentado, no entanto, acontece que em alguns níveis o número de jogadas até à solução é fixo, mesmo havendo várias soluções.

**Conclusão**

A escolha do algoritmo de pesquisa utilizado tem de ser ponderada e avaliada perante o parâmetro que pretendemos otimizar.

Se o interesse do utilizador for rapidez, então o algoritmo DFS ou Greedy será uma boa escolha e até certo nível escalável, sendo que para níveis onde os tabuleiros de jogo são bem maiores mantém na mesma um tempo de resposta rápido.

Se o interesse residir na capacidade de terminar o jogo no menor número de jogadas possíveis, então, nesse caso A\* apresenta-se como uma boa opção, garantindo sempre o menor número de jogadas.

O algoritmo por nós implementado apresentou resultado interessantes, tendo sido o único a conseguir completar todos os níveis em tempos definidos e com rapidez, garantindo sempre a solução ótima.

**Referências**

* GeeksforGeeks:
  + - <https://www.geeksforgeeks.org/greedy-algorithms/>
    - <https://www.geeksforgeeks.org/search-algorithms-in-ai/>
  + Red Blob Games:
    - <https://www.redblobgames.com/pathfinding/a-star/introduction.html?fbclid=IwAR0_pdUYGGfwkUHWjwAkKdxsq_A-1HqhF3XDIaMC0Wx95q_WESjoRqykIUU>
    - <http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html>
    - <http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html#the-a-star-algorithm>
  + Slides da Matéria lecionada em aula.