Métodos de Pesquisa Heurística para resolução de Problemas
(Jogo Folding Blocks)

IART 2020 - Turma 3MIEIC08

Nuno Marques
(up201708997)
MIEIEC, FEUP
up201708997@fe.up.pt

Ricardo Ferreira
(up200305418)
MIEIC, FEUP
ee03195@fe.up.pt

***Abstract*—Neste relatório será apresentado o trabalho a realizado no âmbito da disciplina Inteligência Artificial sobre métodos de pesquisa Heurística. Neste trabalho foi implementada uma versão do jogo folding blocks, cujo objectivo é preencher um tabuleiro de jogo duplicando as formas das peças existentes. O objectivo principal era o desenvolvimento de vários algoritmos de pesquisa que permitam resolver este problema. Estes algoritmos serão comparados a nível de efeciência temporal e espacial para se percerber qual o mais adequado para diferentes problemas. A linguagem escolhida para a implementação foi Java.**

I. Especificação do trabalho a realizar

Este trabalho tem como objectivo desenvolver uma versão do jogo folding blocks [3]. Para além da dinâmica de jogo, a aplicação a desenvolver terá alguns mecânismos de inteligência artificial para resolver os diferentes tabuleiros. Estão previstos dois modos de jogo distintos:

- Computador resolve sozinho
- Humano resolve com possibilidade de pedir dicas ao computador

Para a resolução dos tabuleiros a aplicação deverá implementar diversas técnicas de pesquisa:

- Pesquisa não informada
  - Primeiro em Largura
  - Primeira em profundidade
  - Aprofundamento progressivo
  - Custo uniforme
- Pesquisa Heurística
  - Gulosa
  - A* (com diferentes funções heurísticas)

Os métodos aplicados devem ser comparados para análise da qualidade da solução obtida, número de movimentos e tempo dispendido a procurar uma solução.

A. Descrição do jogo Folding blocks

Folding blocks é um jogo que consiste em preencher um tabuleiro com blocos. O tabuleiro incialmente terá apenas um bloco que pode ser duplicado para formar um novo bloco e posteriormente duplicado novamente até o tabuleiro ficar completamente preenchido. Quando um bloco é duplicado, resulta num novo objecto e é esse novo objecto que pode ser novamente duplicado. Para uma melhor compreensão segue um exemplo nas imagens seguinte:

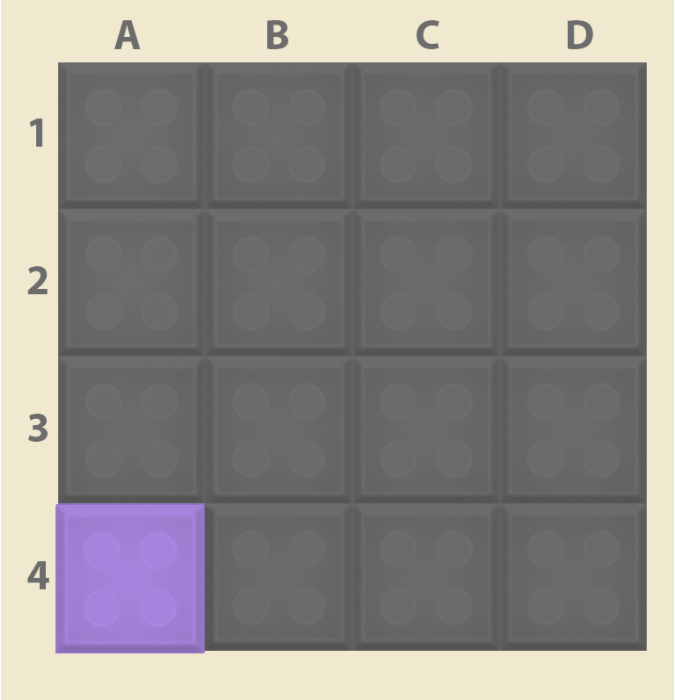

Fig. 1. Exemplo de um tabuleiro de jogo inicial

A figura 1 mostra o estado inicial de um possível jogo, um tabuleiro vazio apenas com um bloco na célula A4. O objectivo será preencher todo o tabuleiro, para isso podemos duplicar o bloco existente formando um rectângulo entre as céclulas A4 e A3 ou A4 e B4. A segunda opção está representada na figura 2.

No estado da figura 2, a peça do jogo é o rectangulo A4-B4 e deve ser esta peça a ser duplicada, neste caso, formando um quadrado entre A3 e B4 ou um rectângulo entre A4 e D4, tal

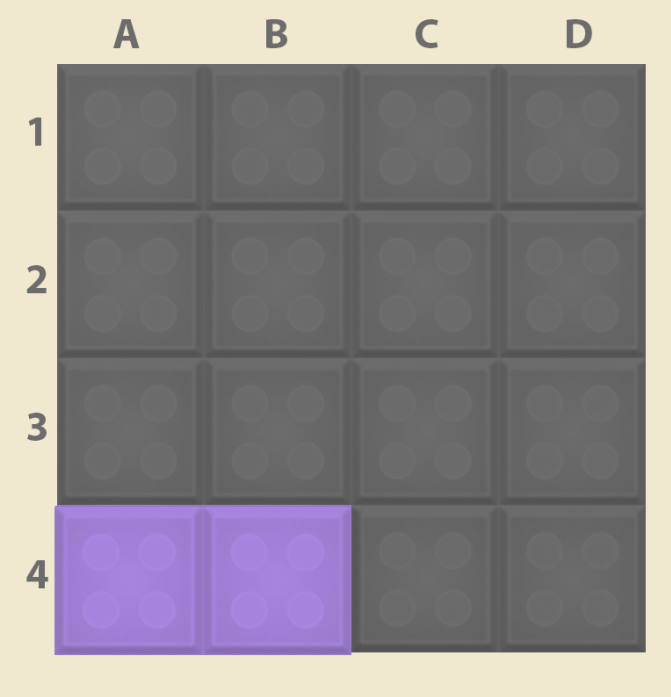

Fig. 2. Exemplo de um tabuleiro de jogo após uma jogada

como mostra a figura 3

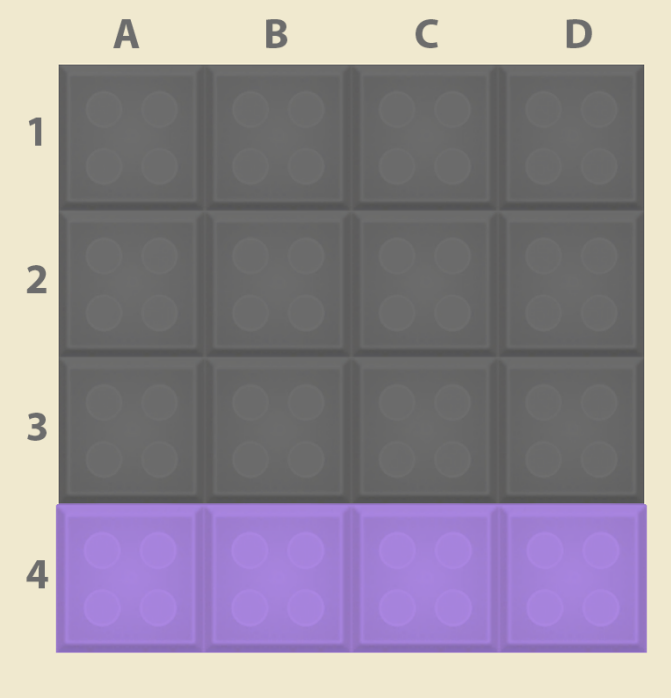

Fig. 3. Exemplo de um tabuleiro de jogo após duas jogada

Continuando a duplicar o objecto resultante é possível preencher todo o tabuleiro e terminar o jogo.

B. Tabuleiros de jogo

Os tabuleiros de jogo podem ter vários formatos, mas nunca com formas arredondadas. Seguem alguns exemplos de tabuleiros. (Figuras 4 até 10)

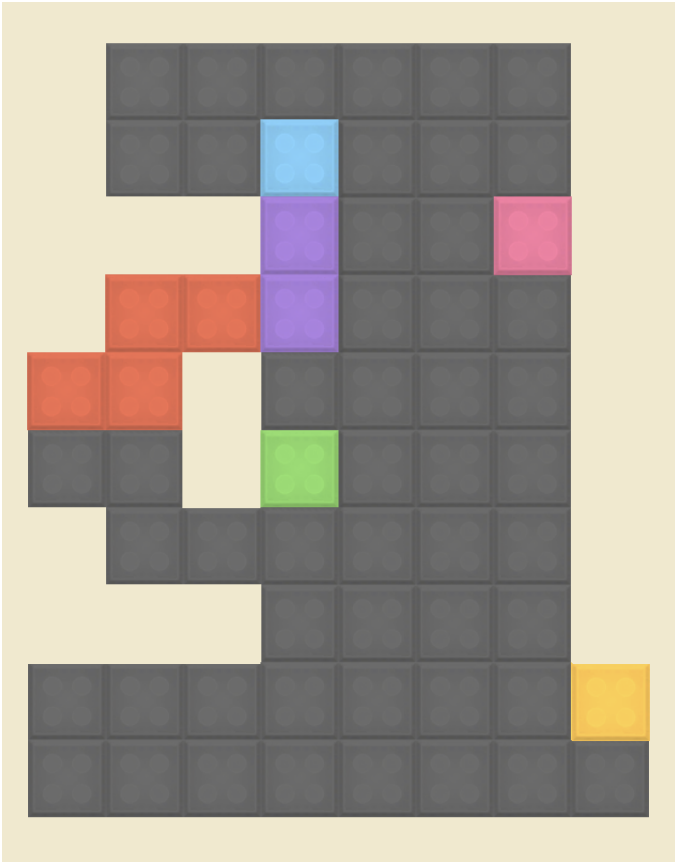

Fig. 4. Exemplo do formato de um tabuleiro de jogo

Os tabuleiros podem começar com uma ou várias peças que podem posteriormente ser duplicadas. Nas figuras 4 até 10 as peças são representadas por cores, mostrando que as peças podem não ter um formato geométrico comum e até estar separadas.

C. Movimentos permitidos

Cada peça no tabuleiro pode ser duplicada formando uma nova peça que posteriormente pode ser duplicada novamente. Para duplicar uma peça, não é permitido sair dos limites do tabuleiro nem duplicar na diagonal. Cada nova peça deve continuar a ter o formato rectangular da peça inicial. É possível duplicar para cima, para baixo, esquerda ou direita, formando assim duas peças simétricas que originaram a peça final.

II. Referências a outros trabahos

Durante a fase de pesquisa foram procurados, artigos e implementações relacionadas com o tema de pesquisa de soluções para jogos. Uma implementação do jogo em análise não foi encontrada, mas implementações relativas a outros puzzles são de igual interesse, visto que abordam as mesmas

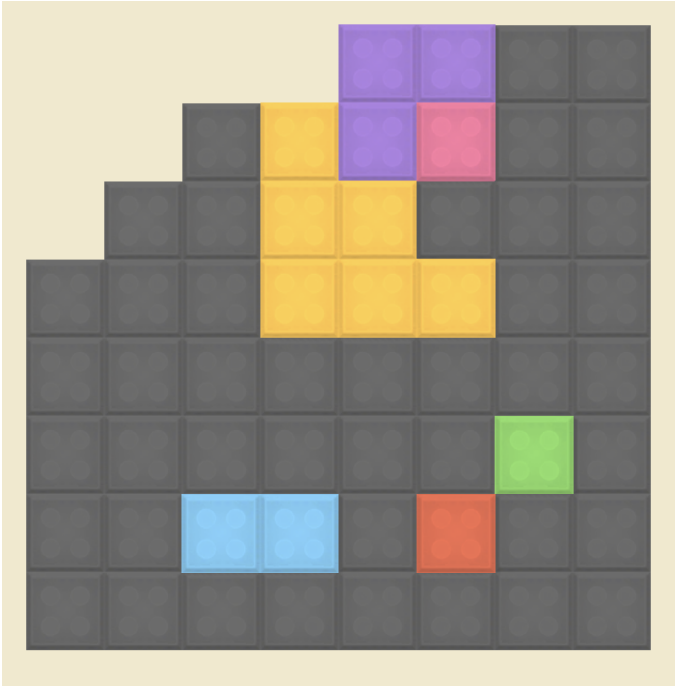

Fig. 5. Exemplo do formato de um tabuleiro de jogo

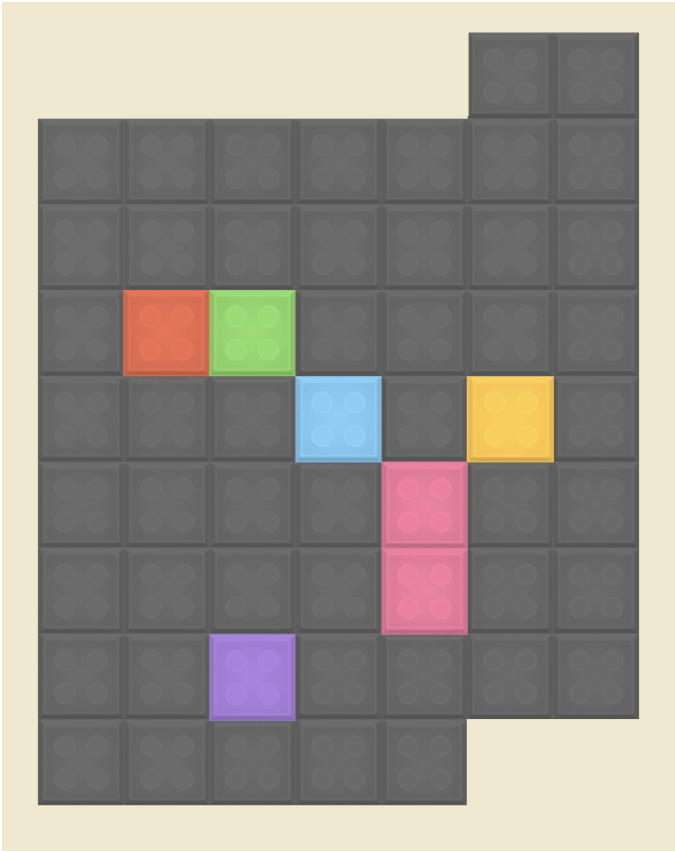

Fig. 6. Exemplo do formato de um tabuleiro de jogo

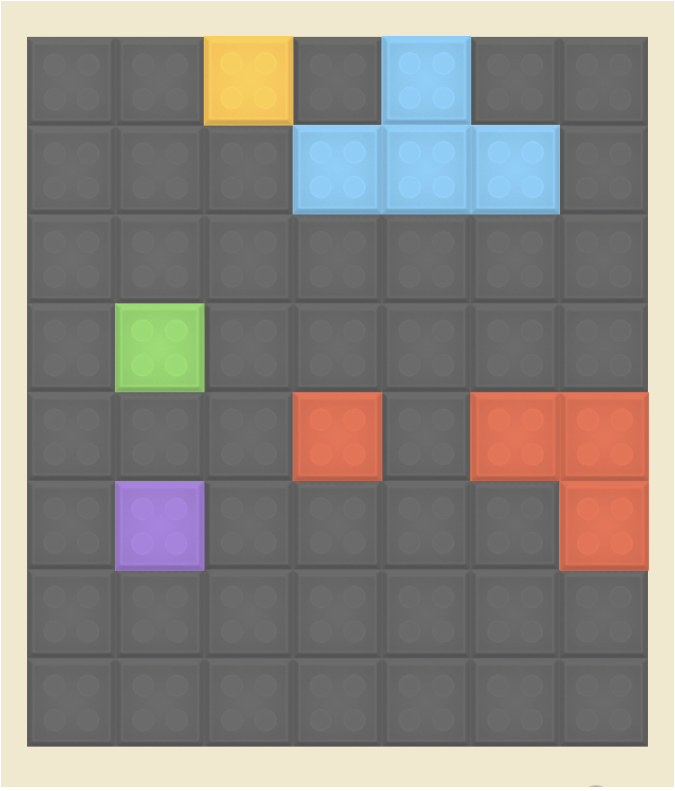

Fig. 7. Exemplo do formato de um tabuleiro de jogo

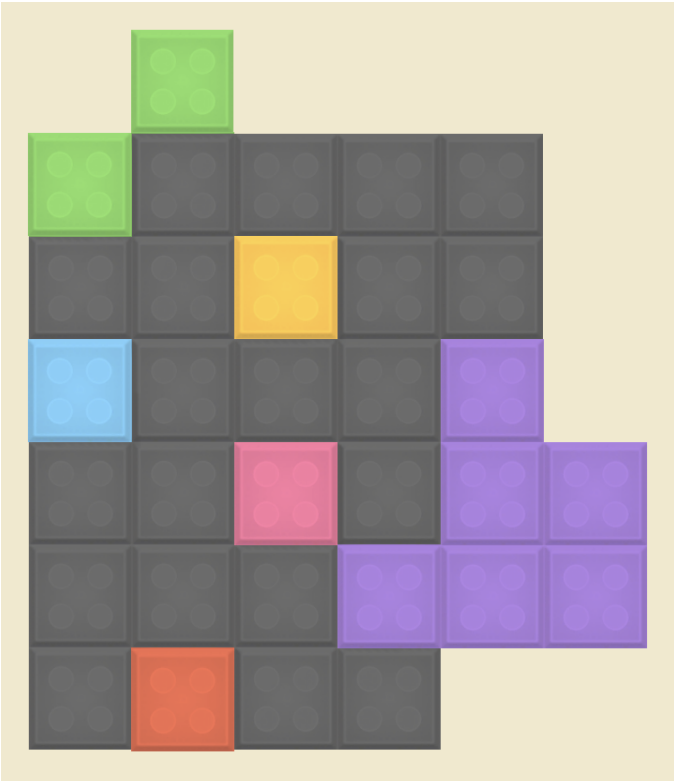

Fig. 8. Exemplo do formato de um tabuleiro de jogo

técnicas de resolução. Segue uma lista e breve descrição das

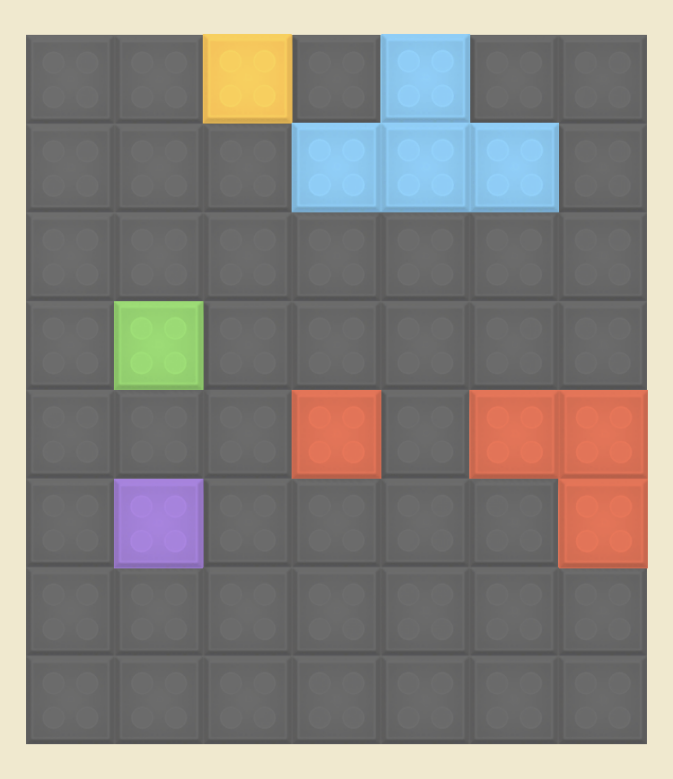

Fig. 9. Exemplo do formato de um tabuleiro de jogo

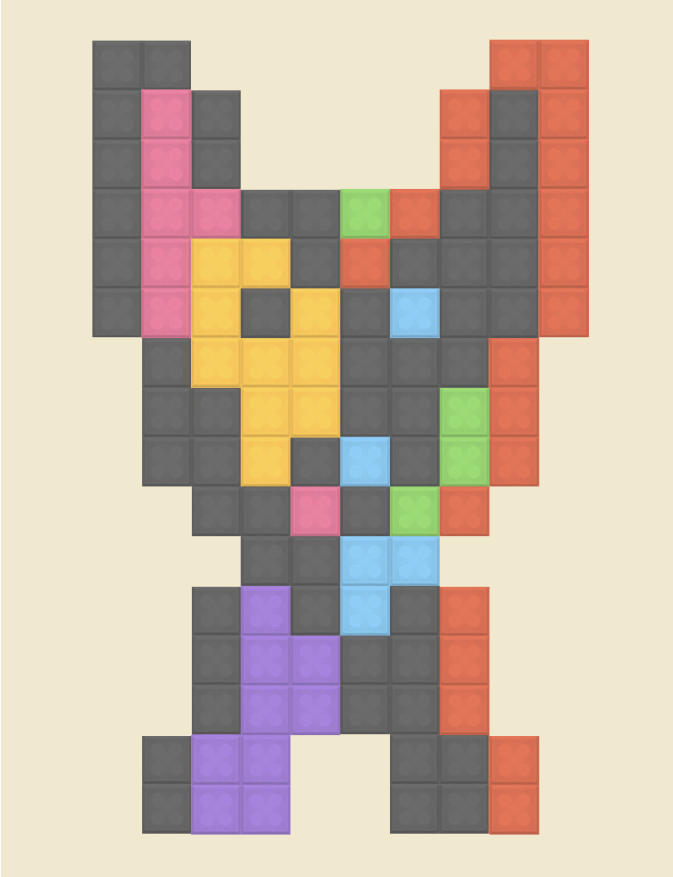

Fig. 10. Exemplo do formato de um tabuleiro de jogo

pesquisas mais relevantes:

- **Using Uninformed Informed Search Algorithms to Solve 8-Puzzle:** Artigo sobre a implementação em Python da resolução do popular n-Puzzle utilizando BFS, DFS, A* e Iterative deepening A*. [1]
- **AI Search Methods:** Repositório de Github com implementações de algoritmos de pesquisa em Java. [2]
- **Searching Algorithms for Artificial Intelligence:** Artigo sobre algoritmos de pesquisa e comparação entre eles. [5]
- **Sliding Puzzle - Solving Search Problem with Iterative Deepening A*:** Artigo em que se formaliza o problema do sliding puzzle e se utiliza aprofundamento progressivo para resolver. [6]
- **Artificial Intelligence: Search** Série de artigos sobre inteligência artificial com análise de uma implementação em python para resolver o n-puzzle utilizando diferentes algoritmos de pesquisa. [4]

III. Formulação do problema

O tabuleiro de jogo será representado como uma matriz nxm com espaços pertences ao jogo e outros não, permitindo assim identificar a área de jogo e criar tabuleiros como os apresentados na secção I-B. As peças de jogo serão representadas por uma lista dos espaços que ocupam no tabuleiro. Cada espaço do tabuleiro terá indicação da posição (coordenada x e y) e variáveis para indicar se pertencem ou não ao jogo e se estão ou não vazios. Convém relembrar que o jogo só é concluído quando todos os espaços de jogo estiverem preenchidos. Os objectos descritos terão a seguinte representação:

```
Board {
    int: Largura n;
    int: Altura m
    List(Block): Lista de peças formadas
no tabuleiro;
    List(Cell): Lista com informação sobre
os espaços do tabuleiro;
}

Block {
    int: identificação da peça
    List(Position): Lista com as posições ocupadas
por esta peça;
}

Cell {
    Position: Posição no tabuleiro;
    boolean: Se pertence ao tabuleiro;
    boolean: Se está vazio;
}

Position {
```

```
    int: Coordenada X
    int: Coordenada Y
}
```

Com esta representação é possível criar um grafo em que cada nó será um estado específico do tabuleiro.

Estado inicial: Será um tabuleiro com uma ou mais peças, de diversas formas, posicionadas em posições específicas. Alguns exemplos de estados iniciais estão representadas na secção I-B.

Teste Objectivo: Verificar se o tabuleiro tem os espaços de jogo todos preenchidos.

Estado final: É o estado de vitória do jogo representado por um tabuleiro com todos os espaços de jogo preenchidos.

A. Operadores

Existem quatro operadores possíveis para cada peça:

- Duplica peça para cima (**DBUP**)
  - **Pré-condição:** Espaços livres na parte de cima da peça
  - **Efeitos:** Formada nova peça por duas partes simétricas
  - **Custo:** 1
- Duplica peça para baixo (**DBDWN**)
  - **Pré-condição:** Espaços livres na parte de baixo da peça
  - **Efeitos:** Formada nova peça por duas partes simétricas
  - **Custo:** 1
- Duplica peça para a esquerda (**DBLFT**)
  - **Pré-condição:** Espaços livres na parte esquerda da peça
  - **Efeitos:** Formada nova peça por duas partes simétricas
  - **Custo:** 1
- Duplica peça para a direita (**DBRGT**)
  - **Pré-condição:** Espaços livres na parte direita da peça
  - **Efeitos:** Formada nova peça por duas partes simétricas
  - **Custo:** 1

Todos os operadores têm custo 1, pelo que o custo total das soluções obtidas será dado pelo número de movimentos necessários para atingir o estado final. O objectivo será alcançar o estado final com o menor número de movimentos e em menor tempo.

IV. Implementação

Este projecto foi desenvolvido utilizando a linguagem de programação JAVA. Sendo uma linguagem orientada a objectos tentou tirar-se partido disso, descrevendo cada elemento do jogo e pesquisa como objectos e criando as respectivas relações de herança e polimorfismo. As estruturas do jogo podem ser divididas em dois módulos: tabuleiro e grafo.

Nas estruturas do tabuleiro existem os objectos já referidos na secção III: *Board*, *Cell*, *Block* e *Position*. Associados a estes objectos foi ainda criada uma interface para gerir as operações permitadas entre estes objectos, chamada *BoardOperations*. Esta interface obriga à implementação das seguintes funções:

- **checkIfPositionsAreAvailable**: verifica se uma lista de positions está vazia no tabuleiro (*Board*)
- **updateBlockWithNewPositions**: adiciona novas posições (*Position*) a um bloco (*Block*), por exemplo quando o bloco é duplicado
- **addBlockAfterValidation**: permite adicionar um bloco (*Block*) ao tabuleiro (*Board*) após verificação se o bloco já existe ou não.
- **validateOccupiedPositions**: permite validar se as peças de um bloco (*Block*) não irão ocupar peças já ocupadas por outros blocos
- **validateBoundaries**: permite validar se as peças de um bloco (*Block*) não ocupam peças fora da área de jogo.

As duas últimas funções são particularmente interessantes pois servirão para criar heuristícas para o algoritmo A* através da relaxação de regras, como será visto na próxima secção.

Neste módulo foi ainda incluído uma classe de enumerados, *Operators* que implementa os operadores apresentados na secção III. Cada um dos operadores possui uma implementação da função *getSymmetricBlockPositions* que retorna uma lista das posições simétricas ocupadas por uma peça se for executado o operador. Cada operador utiliza a função *checkIfPositionsAreAvailable* para validar se as posições calculadas são válidas.

Nas estruturas do grafo existem as estruturas: *Edge* que representam as arestas do grafo com indicação do nó de destino; *Vertex* que representam vértices do grafo, com lista de arestas para nós filhos; *Graph* que representa o grafo com uma lista de arestas. Existe também uma interface para regular as operações entre estas estruturas chamada *GraphOperations*. Esta interface define as seguintes funções:

- **reachObjective**: Verifica se foi atingido o estado final
- **expandGraph**: serve para visitar um determinado nó do grafo e descobrir os nós adjacentes, obdecendo a certas regras expecíficas de cada implementação.
- **addVertexToGraph**: Adiciona um novo nó ao grafo
- **getShortestPath**: Encontra o caminho mais curto entre dois nós

Existe ainda um módulo de search onde se encontram os algoritmos de pesquisa implementados e que serão apresentados com maior pormenor na secção seguinte. Todos os métodos de pesquisa extendem da classe *TraversalStrategy* onde se definem os seguintes métodos:

- **getShortestPath**: Serve como uma simplificação para utilizar as pesquisas, chamando a mesmo função definida em *GraphOperations*. Pode no entanto ser extendida para incluir mais funcionalidades como impressão do caminho por exemplo.
- **getResultNode**: Utilizada para devolver o nó de destino ou de sucesso aplicando o algoritmo de pesquisa.

A. Interação com o utilizador

Com o auxílio das estruturas apresentadas anteriormente a aplicação cria outras estruturas como *Level* para definir os diferentes níveis, *Game* para criar um jogo com as escolhas feitas pelo utilizador e *GameStrategy* para construir um novo jogo de forma interativa mostrando diferentes menus ao utilizador.

Quando se inicia a aplicação o jogador terá vários menus em que poderá seleccionar as seguintes opções:

- Qual a dificuldade pretendida e consequente o nível que irá jogar. Existem 5 níveis diferentes.
- Qual a estratégia de pesquisa a utilizar. Se for seleccionada a pesquisa A* poderá ainda ser seleccionado o tipo de heurística a utilizar, existem 3 diferentes.
- Escolher o modo de jogo, Humano ou AI.

No final é apresentado o resumo das escolhas feitas e a possibilidade de alterar alguma ou todas as opções escolhidas.

Toda a aplicação funciona no terminal. Os menus têm o seguinte formato:

```
Select the level difficulty
====================================
1 - Warm Up
2 - Easy
3 - Medium
4 - Hard
5 - Expert
```

Um tabuleiro de jogo é representado da seguinte forma:

```
+---+---+---+---+---+---+
| & | 1 | & | & | & | & |
+---+---+---+---+---+---+
| 1 |   |   |   |   | & |
+---+---+---+---+---+---+
|   |   | 2 |   |   | & |
+---+---+---+---+---+---+
| 3 |   |   |   | 5 | & |
+---+---+---+---+---+---+
|   |   | 4 |   | 5 | 5 |
+---+---+---+---+---+---+
|   |   |   | 5 | 5 | 5 |
+---+---+---+---+---+---+
|   | 6 |   |   | & | & |
+---+---+---+---+---+---+
```

onde os números identificam as células ocupadas por cada peça e as células com o carácter *&* representam células fora da área de jogo. As restantes células são células vazias.

B. Algoritmos de pesquisa

Todos os algoritmos propostos foram implementados: *pesquisa primeiro em largura*, *pesquisa primeiro em profundidade*, *custo uniforme*, *aprofundamento progressivo*, *guloso* e *A**.

Todos os algoritmos durante a sua execução criam uma lista de estados já visitados, para evitar que o mesmo estado seja explorado duas vezes evitando que se formem ciclos. O algoritmo A* apresenta uma implementação ligeiramente diferente desta lista, pois para além do nó explorado guarda também o custo total calculado para chegar ao nó. Com esta informação extra é possível explorar o mesmo nó desde que este apresente um custo inferior ao já existente na lista, garantindo-se assim que não se descarta um caminho que possa levar a uma solução melhor.

A implementação dos algoritmos *pesquisa primeiro em largura* e *pesquisa primeiro em profundidade* é muito semelhante, mudando apenas o tipo de estrutura utilizada para se guardar o próximo nó a explorar. A pesquisa em profundidade utiliza uma pilha (Stack nas estruturas de java) e a pesquisa em largura uma fila (Queue nas estruturas de java).

O algoritmo de *aprofundamento progressivo* executa uma pesquisa em profundidade limitada, ou seja, o grafo é expandido de forma iterativa, um nível de cada vez e executada pesquisa em profundidade em cada nível de expansão.

O algoritmo de *custo uniforme* expande o grafo e guarda o custo para chegar até determinado nó, utilizando a estrutura *Pair* criada para guardar pares de objectos, neste caso o par, vértice e custo. O custo é calculado pelo número de movimentos necessários para chegar até determinado nó. O algoritmo escolhe o próximo nó a visitar pelo custo mais baixo, existindo no final de cada iteração do algoritmo uma função de ordenação da lista de nós a explorar.

Os algoritmos *guloso* e *A** exploram o grafo aplicando heurísticas para encontrar qual o melhor nó a explorar em cada iteração. As heurísticas são apresentadas na secção seguinte.

C. Heurísticas

Para calcular o interesse em explorar um determinado nó são utilizadas 4 heurísticas diferentes:

- Número de células ocupadas
- Ocupação de células já ocupadas
- Ocupação de células fora do tabuleiro de jogo
- Ocupação de células já ocupadas e fora do tabuleiro de jogo

A heurística do número de células ocupadas é utilizada em conjunto com o algoritmo *guloso*. Esta heurística calcula o número de células já ocupadas em cada estado, escolhendo-se para explorar o nó com o maior número de células ocupadas. O nó com o maior número de células ocupadas tem maior interesse pois, em princípio, será preciso um menor número de movimentos para preencher as restantes células do tabuleiro.

As restantes heurísticas resultam de um relaxamento das regras de jogo para permitirem obter um custo estimado para se atingir o objectivo e são utilizadas com o algoritmo *A**. Em cada iteração do algoritmo é utilizada a função seguinte para estimar o número de movimentos necessários para atingir o objectivo final.

```
f(n) = g(n) + h(n)
  - f(n): custo estimado da solução
```

Nível Warm Up			
Largura	Tempo(ms): 33 Nós: 9 Movimentos: 5		
Profundidade	Tempo(ms): 26 Nós: 7 Movimentos: 5		
Aprofundamento	Tempo(ms): 40 Nós: 9 Movimentos: 5		
Custo Uniforme	Tempo(ms): 51 Nós: 9 Movimentos: 5		
Gulosa	Tempo(ms): 32 Nós: 7 Movimentos: 5		
A*	**Ocupadas**	**Fora Tabuleiro**	**Sem Regras**
	Tempo(ms): 49 Nós: 9 Movimentos: 5	Tempo(ms): 38 Nós: 7 Movimentos: 5	Tempo(ms): 36 Nós: 7 Movimentos: 5

TABLE I
RESULTADOS DOS ALGORITMOS NO NÍVEL WARM-UP.

Nível Easy			
Largura	Tempo(ms): 353 Nós: 162 Movimentos: 10		
Profundidade	Tempo(ms): 58 Nós: 30 Movimentos: 10		
Aprofundamento	Tempo(ms): 722 Nós: 162 Movimentos: 10		
Custo Uniforme	Tempo(ms): 1323 Nós: 162 Movimentos: 10		
Gulosa	Tempo(ms): 58 Nós: 30 Movimentos: 10		
A*	**Ocupadas**	**Fora Tabuleiro**	**Sem Regras**
	Tempo(ms): 1092 Nós: 121 Movimentos: 10	Tempo(ms): 146 Nós: 26 Movimentos: 10	Tempo(ms): 649 Nós: 95 Movimentos: 10

TABLE II
RESULTADOS DOS ALGORITMOS NO NÍVEL EASY.

```
- g(n): custo total até ao momento
- h(n): custo estimado para chegar
ao objectivo (optimista)
```

A heurística de ocupação de células já ocupadas, faz um cálculo de h(n) permitindo que um bloco possa ser duplicado para posições já ocupadas por outros blocos, sem sair do tabuleiro de jogo. Desta forma é possível gerar blocos maiores em menos jogadas e atingir o objectivo de forma mais rápida.

A heurística de ocupação de células fora do tabuleiro, faz um cálculo de h(n) permitindo que um bloco possa ser duplicado para posições que não pertencem ao tabuleiro de jogo, mais uma vez permitindo blocos de maiores dimenções e como tal menor número de jogadas para atingir o objectivo.

A heurística de ocupação de células já ocupadas e fora do tabuleiro, combina as duas eurísticas anteriores, ou seja, os blocos podem ser expandidos sem regras.

Em situações de empate, ou seja, estados com o mesmo valor de custo estimado, utiliza-se a mesma heurística utilizada no algoritmo guloso para escolher o melhor movimento seguinte.

V. RESULTADOS

Foram realizados vários testes com 5 níveis diferentes: Warm-Up, Easy, Medium, Hard, Expert.

Os resultados obtidos em cada um dos níveis está representado nas tabelas I até V.

No nível extreme o método A* com a heurística de ocupação de células ocupadas não conseguiu produzir um resultado em tempo útil. Isto deve-se ao facto de nesta estratégia as células terem sempre movimentos possíveis, uma vez que podem ocupar as posições de outra células, como tal, o espaço de estados a explorar é muito grande. Seria de esperar um resultado semelhante na heurística sem regras, mas tal não se verifica pois, como as peças podem ir para áreas não pertencentes ao tabuleiro as peças acabam por ocupar áreas

Nível Medium			
Largura	Tempo(ms): 265 Nós: 108 Movimentos: 10		
Profundidade	Tempo(ms): 48 Nós: 24 Movimentos: 10		
Aprofundamento	Tempo(ms): 391 Nós: 108 Movimentos: 10		
Custo Uniforme	Tempo(ms): 728 Nós: 108 Movimentos: 10		
Gulosa	Tempo(ms): 60 Nós: 23 Movimentos: 10		
A*	**Ocupadas**	**Fora Tabuleiro**	**Sem Regras**
	Tempo(ms): 1010 Nós: 95 Movimentos: 10	Tempo(ms): 183 Nós: 23 Movimentos: 10	Tempo(ms): 331 Nós: 25 Movimentos: 10

TABLE III
RESULTADOS DOS ALGORITMOS NO NÍVEL MEDIUM.

Nível Hard			
Largura	Tempo(ms): 168 Nós: 54 Movimentos: 8		
Profundidade	Tempo(ms): 56 Nós: 20 Movimentos: 8		
Aprofundamento	Tempo(ms): 222 Nós: 54 Movimentos: 8		
Custo Uniforme	Tempo(ms): 362 Nós: 54 Movimentos: 8		
Gulosa	Tempo(ms): 52 Nós: 20 Movimentos: 8		
A*	**Ocupadas**	**Fora Tabuleiro**	**Sem Regras**
	Tempo: 687 Nós: 54 Movimentos: 8	Tempo: 119 Nós: 17 Movimentos: 8	Tempo: 442 Nós: 53 Movimentos: 8

TABLE IV
RESULTADOS DOS ALGORITMOS NO NÍVEL HARD.

Nível Expert			
Largura	Tempo(ms): 11809 Nós: 779 Movimentos: 8		
Profundidade	Tempo(ms): 68 Nós: 42 Movimentos: 8		
Aprofundamento	Tempo(ms): 21264 Nós: 779 Movimentos: 8		
Custo Uniforme	Tempo(ms): 57842 Nós: 779 Movimentos: 8		
Gulosa	Tempo(ms): 73 Nós: 38 Movimentos: 8		
A*	**Ocupadas**	**Fora Tabuleiro**	**Sem Regras**
	Tempo: NA Nós: NA Movimentos: NA	Tempo: 44019 Nós: 36 Movimentos: 8	Tempo: 7147 Nós: 263 Movimentos: 8

TABLE V
RESULTADOS DOS ALGORITMOS NO NÍVEL EXPERT.

maiores e terminar o jogo em menos jogadas, logo o espaço a explorar é mais reduzido.

Os gráficos das figuras 11 até 15 mostram uma comparação dos tempos de execução, para permitir uma melhor análise dos resultados.

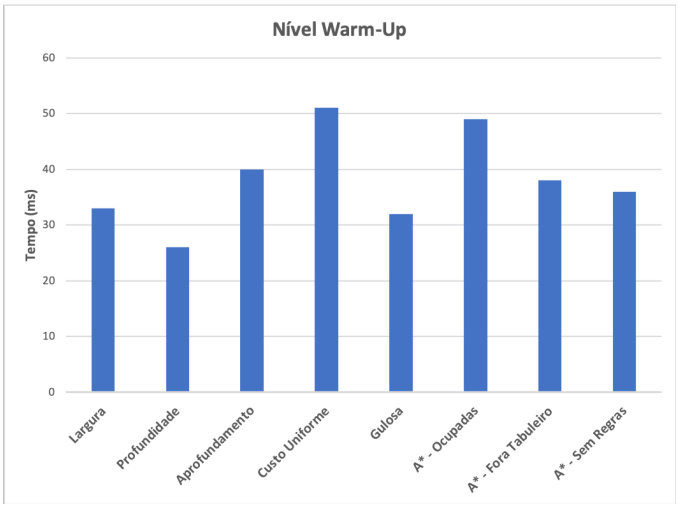

Fig. 11. Relação de tempos de execução no nível Warm-Up

É possível observar que os algoritmos de pesquisa em profundidade e pesquisa gulosa são os que apresentam melhores resultados. Isto deve-se ao facto de este jogo apenas ter uma solução possível e como tal apenas um caminho para chegar à solução final. Assim, a pesquisa em profundidade quando encontra o nó com o caminho correcto e o começa a explorar rapidamente chega a um resultado final, por sua vez a pesquisa gulosa como preveligia o nó com mais posições ocupadas também encontra o caminho correcto logo nos primeiros níveis de exploração, não perdendo tempo a explorar nós que não levam ao resultado.

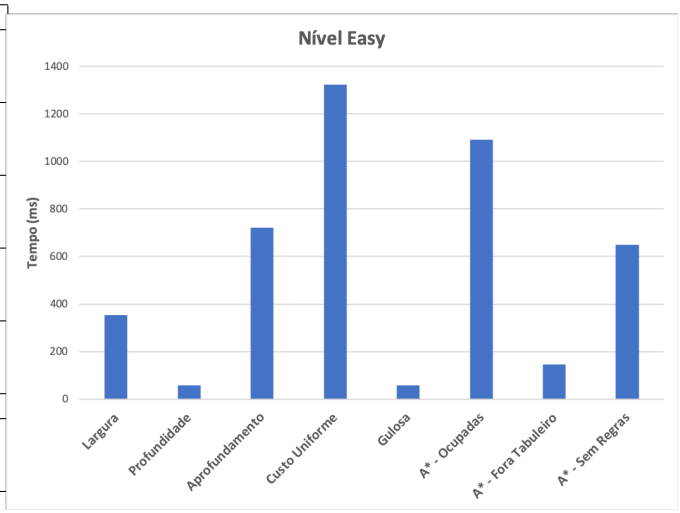

Fig. 12. Relação de tempos de execução no nível Easy

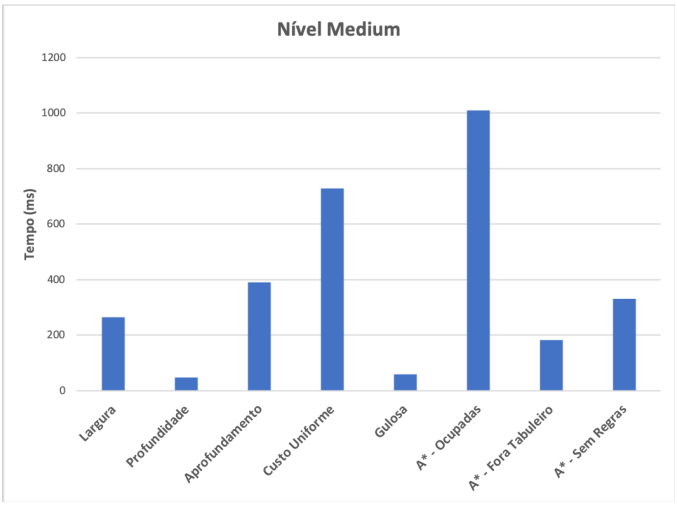

Fig. 13. Relação de tempos de execução no nível Medium

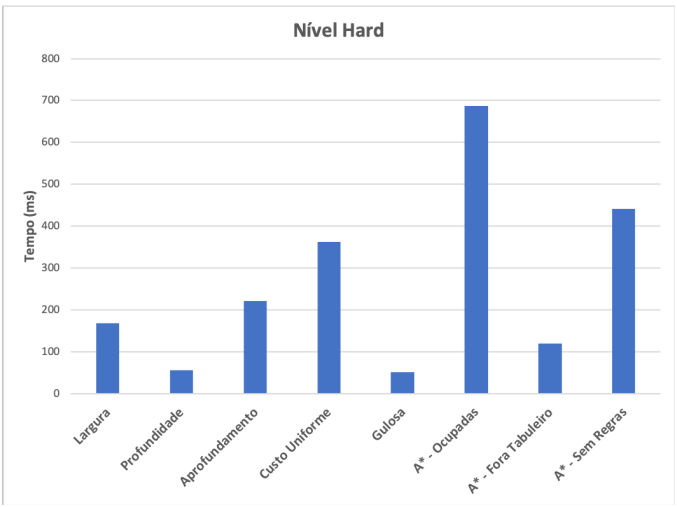

Fig. 14. Relação de tempos de execução no nível Hard

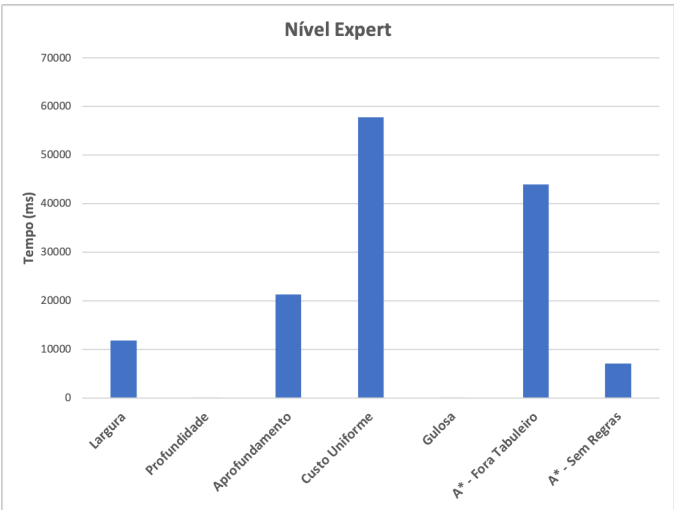

Fig. 15. Relação de tempos de execução no nível Expert

VI. Conclusão

O trabalho implementado permitiu comparar os diferentes algoritmos de pesquisa para resolução do jogo folding blocks. Devido à forma como o jogo é jogado, em que apenas existe uma forma de resolver os tabuleiros, todos os algoritmos apresentam o mesmo número de movimentos. Podemos então concluir que uma heurística baseada no número de movimentos não é a mais adequada para este caso. Isto explica os maus resultados do algoritmo A*. A heuristica mais adequada de resolução é a heuristica que tenta ocupar o máximo de posições possíveis em cada jogada, o que se pode verificar pelos resultados obtidos pelo método de pesquisa gulosa. Nos métodos de pesquisa não informada, o algoritmo de pesquisa primeiro em profundidade apresenta os melhores resultados, novamente pode-se explicar este comportamente pela natureza do jogo. Este método como pesquisa os nós em profundidade vai chegar rapidamente ao resultado final, pois para cada tabuleiro só existe uma forma de o resolver, quando o algoritmo começa a explorar o nó que conduz ao resultado final rápidamente o atinge, ao contrário dos outros métodos que tentam verificar outros nós antes de aprofundarem o nó actual. Devido ás heuristicas utilizadas, baseadas no número de movimentos, o algoritmo A* apresenta um mau desempenho, principalmente em níveis mais complexos. Isto deve-se ao facto das heurísticas utilizadas conduzirem a uma grande expansão de nós o que aumenta consideravelmente o espaço e a memória utilizada por este método. Isto verifica-se principalmente na heurística de ocupação de células já ocupadas que faz com que haja sempre movimentos possíveis para todas as peças em todos os níveis de exploração, levando a um número de nós a explorar demasiado elevado. O mesmo comportamente verifica-se no algoritmo de custo uniforme, pois cada nível de exploração tem custo sempre 1, o que faz com que este algoritmo visite todos os nós até chegar á solução final. Finalmente, podemos concluir que os métodos de pesquisa são eficazes se utilizados com as heurísticas adequadas para cada caso de uso e que nem todos os algoritmos se podem adaptar a todos os usos de caso, para o jogo folding blocks o algoritmo de pesquisa primeiro em profundidade ou guloso é o mais adequado.

References

[1] Sandipan Dei. Using uninformed informed search algorithms to solve 8-puzzle (n-puzzle) in python. URL: https://www.datasciencecentral.com/profiles/blogs/using-uninformed-informed-search-algorithms-to-solve-8-puzzle-n, 7 June 2017. [Online; acedido em 15-Março-2020].

[2] Sandipan Dei. Using uninformed informed search algorithms to solve 8-puzzle (n-puzzle) in python. URL: https://www.datasciencecentral.com/profiles/blogs/using-uninformed-informed-search-algorithms-to-solve-8-puzzle-n, 7 June 2017. [Online; acedido em 15-Março-2020].

[3] Popcore Games. Folding blocks: Puzzle game. URL: https://play.google.com/store/apps/details?id=com.popcore.foldingblocks. [Online; acedido em 15-Março-2020].

[4] Will Koehrsen. Artificial intelligence: Search*. URL: https://medium.com/@williamkoehrsen/artificial-intelligence-part-1-search-a1667a5991e5, 22 Sep 2017. [Online; acedido em 15-Março-2020].

[5] Made lapuerta. Sliding puzzle - solving search problem with iterative deepening a*. URL: https://medium.com/datadriveninvestor/searching-algorithms-for-artificial-intelligence-85d58a8e4a42, 18 Jan 2019. [Online; acedido em 15-Março-2020].

[6] Greg Surma. Searching algorithms for artificial intelligence. URL: https://medium.com/datadriveninvestor/searching-algorithms-for-artificial-intelligence-85d58a8e4a42, 25 Sep 2019. [Online; acedido em 15-Março-2020].