Resolução do jogo Break the Ice utilizando métodos de pesquisa em linguagem Java (Tema 2 Grupo 5)

1st Eduardo Silva (up201603135)
MIEIC
FEUP
Porto, Portugal
up201603135@fe.up.pt

2nd Mariana Costa (up201604414)
MIEIC
FEUP
Porto, Portugal
up201604414@fe.up.pt

3rd Tiago Castro (up201606186)
MIEIC
FEUP
Porto, Portugal
up201606186@fe.up.pt

***Resumo*—O presente artigo tem como principal objetivo delinear os contornos de desenvolvimento da resolução do jogo Break the Ice utilizando métodos de pesquisa em linguagem Java. Será realizada um descrição sucinta do jogo em questão bem como da sua implementação, seguida da explicação dos vários algoritmos de pesquisa usados (largura, profundidade, profundidade iterativa, custo uniforme, guloso, A*).**

***Index Terms*—Inteligência Artificial, Pesquisa, Jogo, Break the Ice, Largura, Profundidade, A*, Guloso, Custo Uniforme, Profundidade Iterativa**

I. Introdução

O projeto aqui exposto insere-se no âmbito da unidade curricular de Inteligência Artificial do MIEIC. É explicada a resolução do Break the Ice, um jogo do tipo solitário, utilizando métodos de pesquisa adequados em linguagem Java. O artigo segue a estrutura IEEE e é possível consultar as várias secções do mesmo para obter informação pormenorizada à secção em questão.

II. Descrição do problema

Break The Ice: Snow World [1] é um jogo de azulejos no qual o objetivo principal consiste em mover as peças em jogo (azulejos) para um espaço vazio ou trocá-las com peças adjacentes de modo a criar uma cadeia vertical ou horizontal de três ou mais peças. De cada vez que estas cadeias são criadas, as peças a elas pertencentes são quebradas, consequentemente desaparecendo do tabuleiro de jogo. O nível é ganho quando todas as peças deste são quebradas, ou seja, quando não restar nenhuma peça. No entanto, é de notar que há um número limitado de jogadas (movimentos) por nível, sendo que o jogador perde se ultrapassar este limite por mais de 2 (i.e. num nível com 3 movimentos possíveis, completá-lo com 5 jogadas resulta em 1 estrela de 3, 4 em 2 e 3 em 3).

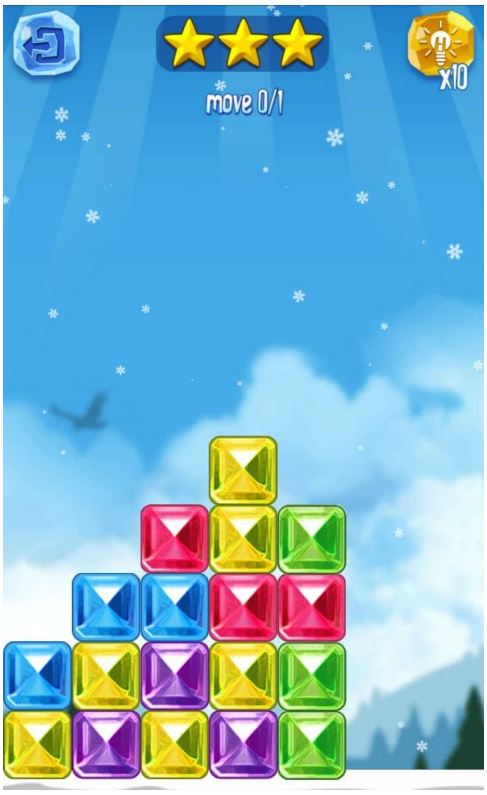

Figura 1. Ecrã inicial de um puzzle

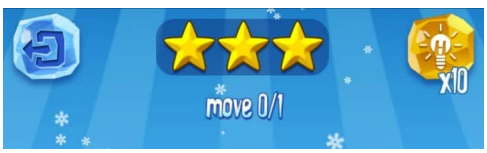

Figura 2. Sistema de custo por jogada

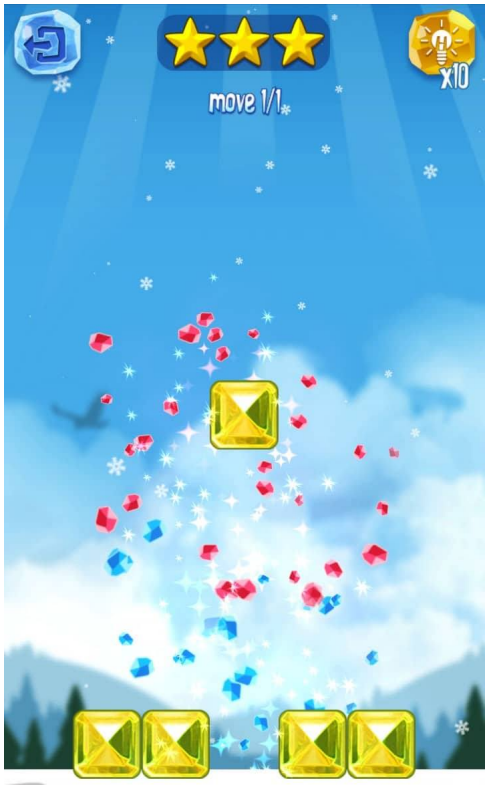

Figura 3. Movimento das peças

III. Formulação do problema

A. Representação do estado

Estados representados por uma matriz 2D de carateres em que cada elemento representa um espaço vazio ou um elemento (peça). A peças de diferentes cores irão corresponder carateres diferentes, assim como aos espaços vazios. As peças podem ser roxas, laranjas, rosas, azuis, verdes e amarelas.

B. Estados iniciais

Peças distribuídas pela matriz, sendo que não há nenhum conjunto de 3 ou mais peças da mesma cor alinhadas ortogonalmente.

C. Teste objetivo

Para um nível ser considerado completo, todas as peças têm que ter sido eliminadas, ou seja, a matriz contém apenas espaços vazios.

D. Operadores

- *Mover peça (esquerda, direita)*
  - **Pré-condições**: Existir um peça na posição selecionada e a posição de destino não corresponder a uma posição fora da matriz de jogo.
  - **Efeitos**: A peça move-se uma posição para a esquerda ou direita. Caso a peça esteja inicialmente numa posição superior à última linha da matriz e na posição para a qual se tenha movido não haja outro elemento na posição imediatamente inferior, a peça percorre as linhas da matriz por ordem crescente de índice até encontrar outro elemento ou até atingir a última linha na matriz.
  - **Custo**: 1 jogada
- *Trocar peças (esquerda, direita, cima, baixo)*
  - **Pré-condições**: As peças a trocar devem estar em posições adjacentes ortogonais.
  - **Efeitos**: As peças trocam de posição.
  - **Custo**: 1 jogada

E. Efeitos gerais

Se após uma destas operações houver pelo menos 3 peças da mesma cor em linha (ortogonalmente), todas as peças dessa cor que estejam em contacto com eles irão desaparecer (para além do conjunto inicial).

F. Custo da solução

Cada movimento tem um custo de 1 jogada, sendo o custo total da solução o número de movimentos realizados para resolver o problema.

IV. Trabalho relacionado

Durante a pesquisa para este trabalho, foi encontrado um projeto [2] com um objetivo semelhante num contexto bastante parecido ao presente. Este projeto retrata um jogador AI para uma versão clone do popular jogo Candy Crush. Este jogo é, em si, bastante parecido ao puzzle central deste trabalho. Além disso, esse projeto foi realizado na mesma linguagem que o presente trabalho (Java), portanto poderá haver módulos úteis em que nos possamos inspirar para a conclusão deste.
Foi encontrado também um relatório [3] que analisa multiplos métodos de busca para resolução de puzzles do jogo Candy Crush sendo alguns deles a pesquisa em largura e a pesquisa gulosa.

V. Implementação do jogo

Segue-se uma descrição dos principais pormenores da implementação do projeto na linguagem escolhida (Java), dividida nas secções consideradas mais relevantes para uma compreensão robusta do trabalho realizado. De notar que apenas os métodos avaliados como estando a carecer de aprofundamento ou cujo propósito não é de compreensão imediata serão explicados em detalhe.

A. Estrutura de ficheiros

O programa encontra-se repartido por três *packages*, cada um com responsabilidades distintas:

- ***game***: Contém toda a lógica de jogo, bem como as classes responsáveis pela resolução do problema com recurso aos diferentes métodos de pesquisa;
- ***heuristic***: Contém as classes representativas das várias heurísticas utilizadas pelos algoritmos de pesquisa informada (baseado na estrutura de ficheiro do projeto mencionado nos trabalhos relacionados [3]);
- ***testing***: Contém as classes utilizadas para o teste autónomo da resolução dos níveis do jogo e a avaliação do desempenho dos algoritmos implementados;

B. Representação do estado do tabuleiro

Um tabuleiro de jogo é representado pela classe *GameBoard*, que contém a informação necessária à avaliação do mesmo:

- ***char[][] board***: Matriz de *chars*, na qual cada posição pode estar vazia ou conter uma peça válida, representada por um *char* pertencente ao conjunto acima referido;
- ***int maxMoves***: Atributo que reflete o número máximo de movimentos da board em questão;
- ***boolean showMove***: Valor utilizado para determinar se, durante a movimentação de uma peça no tabuleiro, este deve ser impresso na consola ou não;

C. Operadores

Os operadores de jogo presentes na classe *GameBoard* procuram traduzir todos os movimentos aceites pela lógica de jogo. Tendo em conta que os movimentos válidos consistem unicamente na translação de peças no tabuleiro, quer por troca quer por deslocação para um determinado espaço vazio, todos eles seguem uma estrutura semelhante, variando apenas na direção do movimento e consequentemente das posições analisadas para a validação do mesmo.

Assim, os principais métodos utilizados para a concretização de uma jogada possuem a designação de *movePiece*[Direção] ou *switchPiece*[Direção], aceitando como argumento um *int[]* contendo as coordenadas da peça a movimentar-se e retornando um objeto do tipo *GameBoard* cujo atributo *board* possui o tabuleiro de jogo atualizado.

Para a validação das jogadas é utilizado o método *validateMove*[Direção], que determina se, após a jogada, as peças se encontram dentro dos limites do tabuleiro (ou seja, se as suas coordenadas se traduzem em índices válidos da matriz) e se as pré-condições do operadores são respeitadas. Aceita como argumento as coordenadas da peça a movimentar-se e retorna um booleano que reflete o resultado da análise realizada (*true* se a deslocação for válida, *false* caso contrário).

D. Representação do custo e efeitos das jogadas

Os efeitos de cada operador são concretizados nos métodos *dropPiece* e *explode*[...], responsáveis pela aplicação dos efeitos da gravidade caso seja necessário e pela eliminação de sequências de pelo menos 3 peças adjacentes da mesma cor, respetivamente.

E. Execução dos algoritmos

A classe *Bot* encontra-se encarregue de inicializar a pesquisa de soluções consoante o algoritmo selecionado pelo utilizador no menu (bem como a heurística, caso esta se aplique), nomeadamente através do método *search*. Os métodos relativos aos algoritmos (*breadth*, *greedySearch*, *uniformCostSearch* e *AStarSearch* e o método auxiliar *aux*) irão ser detalhados na secção seguinte.

F. Listagem de jogadas possíveis e análise da viabilidade de um tabuleiro

A listagem de todas as jogadas possíveis num dado tabuleiro é realizada pelo método *expandNode* da classe *GameNode*, que retorna uma *ArrayList* de todos os objetos do tipo *GameNode* descendentes do objeto *GameNode* a ser analisado.

O método *isImpossibleState* determina se está a ser avaliado um estado a partir do qual não é possível alcançar uma solução (se, por exemplo, o número de blocos de uma só cor for inferior a 3 e superior a 0), impedindo assim a expansão de nós irrelevantes.

G. Heurísticas

De forma a facilitar a implementação de várias heurísticas, foi introduzida a classe abstrata *Heuristic*, cujo atributo *value* reflete o valor da mesma, sendo este utilizado aquando da inserção de elementos na *PriorityQueue* presente nos métodos responsáveis pelos algoritmos de custo uniforme e de pesquisa informada (A* e pesquisa gulosa). Encontram-se na enumeração seguinte as generalizações desta classe, descritas detalhadamente na secção dedicada aos algoritmos de pesquisa: *BlockNumHeuristic*, *CloseChainHeuristic* e *ColorHeuristic*.

Com o intuito de utilizar a *PriorityQueue* com elementos do tipo *Node*, foi concretizado o *overload* ao método *compareTo* para que objetos deste tipo pudessem ser inseridos nesta estrutura de dados.

H. Teste objetivo

A determinação da condição de vitória é realizada pelo método *testGoal* da classe *GameNode*; este verifica se, dada uma matriz de *chars*, as células do tabuleiro se encontram vazias (de notar que o estado 'vazio' corresponde ao char '_'). A chamada a este método é realizada em diferentes momentos durante a pesquisa da solução consoante o algoritmo escolhido.

I. Obtenção da solução

O método *traceSolution* executa a solução obtida após a execução do algoritmo selecionado. A solução consiste num conjunto de *Strings* resultante da concatenação de todos os operadores necessários para completar o nível. Um exemplo de um operador seria: "Switch left (2,2)"

O método *traceSolutionUp* parte do nó objetivo e percorre recursivamente a solução obtida, adicionando o operador de cada nó pelo qual passa a uma *ArrayList* de *Strings* com os operadores todos necessarios para a executar.

J. Visualização do estado de jogo

Para realizar a interpretação visual dos resultados obtidos pelos algoritmos de pesquisa, foi implementado o método *printBoard* na classe *GameBoard*, responsável pela impressão do tabuleiro de jogo na consola.

K. Representação do menu e leitura de ficheiros

A classe *BreakTheIce* possui os métodos necessários à interação entre o utilizador e a máquina, possibilitando a escolha de nível, algoritmo (*chooseMode*) e heurística a utilizar (*chooseHeuristic*) se for aplicável. Na escolha de nível o utilizador pode optar por um dos 6 níveis *default* (método *chooseLevel*) ou carregar um nível a partir de um ficheiro de texto (método *loadLevel*).

L. Funções de teste e análise de resultados

Como referido anteriormente, o *package testing* encontra-se responsável pelas funções utilizadas no estudo dos resultados obtidos na execução dos algoritmos. Para isto, foram concretizadas os seguintes métodos *testLevels* e *randomTest* pertencentes à classe *TestingSuite*. É possível também testar níveis submetidos através de ficheiros.

VI. Algoritmos de pesquisa

Procurando cobrir um espectro variado de métodos utilizados na procura de soluções, foram implementados os algoritmos enumerados de seguida, acompanhados por uma breve descrição teórica bem como pela sua implementação específica ao problema em questão. Segue-se ainda um subsecção com algumas considerações gerais.

A. Considerações gerais

As principais estruturas de dados utilizadas na execução dos algoritmos são *ArrayList* e *PriorityQueue*, sendo relevante detalhar algumas especificidades da implementação desta última, descrição esta feita mais à frente.

Os métodos responsáveis por pesquisas em profundidade (*depthSearch* e *iterativeDeepeningSearch*) foram implementados na classe *GameNode*, enquanto que as pesquisas em largura e pesquisas informadas (*breadth*, *uniformCostSearch*, *greedySearch* e *AStarSearch*) se encontram em métodos da classe *Bot*. Esta discrepância deve-se à utilização (ou ausência dela) de recursividade. Assim, consiste meramente numa forma de facilitar a estruturação do método e, consequentemente, obter um código mais limpo, evitando que os nós a analisar recursivamente necessitem de ser passados por argumento ao método.

Dado que a pesquisa de custo uniforme e pesquisa gulosa partilham um estratégia de resolução semelhante, diferindo apenas na ordem pela qual os nós são avaliados, foi implementado um método auxiliar *aux* utilizado por ambos. A consistência da colocação dos elementos na *PriorityQueue* com o algoritmo em uso é garantida aquando da escolha deste, sendo que a classe *Node* possui um atributo *searchOption*, utilizado pelo método *compareTo* para determinar a estratégia de comparação de objetos do tipo *Node*. Esta função auxiliar percorre uma *PriorityQueue* até que esta se encontre vazia, verificando a cada iteração se o nó atual responde positivamente ao teste objetivo, bem como expandindo o nó e acrescentando os descendentes à estrutura de dados mediante algumas estipulações. Estas impõem que os nós colocados na *PriorityQueue* nao sejam repetidos e que a sua expansão não seja realizada caso a profundidade do nó atual seja igual ou superior ao nível limite de profundidade ditado pelo número máximo de jogadas do nível a ser analisado. A pesquisa A* é também em muito semelhante às duas anteriores; contudo, a sua implementação neste jogo implica algumas verificações adicionais relativamente a nós repetidos, pelo que se encontra numa função distinta das restantes.

B. Pesquisa não informada

1) Pesquisa em largura (Breadth-first search): Na pesquisa em largura, é primeiramente realizada a expansão da raíz, seguida da expansão de todos os nós filhos e assim sucessivamente até que o nó objetivo seja alcançado. Contudo, a sua complexidade exponencial tanto em tempo como em espaço implica um consumo dos recursos acima mencionados muito elevado. Trata-se de um algoritmo completo e ótimo.

Concretamente, é implementado um ciclo com um número de iterações correspondente ao número máximo de jogadas do nível a ser estudado. A cada iteração, a *ArrayList* contém todos os nós do nível de profundidade atual, sendo que cada nó analisado é expandido e os nós resultantes dessa expansão são adicionados a uma segunda *ArrayList* auxiliar que, no final da iteração atual é acrescentada à estrutura de dados de nós ativos.

2) Pesquisa em profundidade (Depth-first search): Na pesquisa em profundidade, é sempre explorado um dos nós mais profundos da árvore; ou seja, é percorrido um ramo até alcançar a profundidade máxima, momento após o qual, caso não seja encontrada uma solução, é realizado *backtracking*. Consiste numa estratégia que requer pouca memória, particularmente relevante na resolução de problemas com muitas soluções. Porém, não pode ser utilizado em árvores de profundidade infinita uma vez que corre o risco de explorar um ramo infindável no qual não existe uma solução. Trata-se de um algoritmo não completo e não ótimo.

Relativamente à sua implementação no projeto, pode ser descrito como uma função recursiva. Caso o nível de profundidade do nó a ser testado de momento seja igual ou superior ao nível de profundidade máximo para o nível, a pesquisa termina de forma a impedir que o programa entre num ciclo infinito.

3) Aprofundamento iterativo (Iterative deepening): O aprofundamento iterativo consiste numa variante da pesquisa em profundidade, na qual esta pesquisa é realizada até um limite de profundidade sucessivamente maior em cada iteração até que o nó objetivo seja alcançado. A complexidade temporal é exponencial, enquanto que a complexidade espacial é linear. Trata-se de uma estratégia completa e ótima, indicada para problemas com um grande espaço de pesquisa e profundidade da solução desconhecida.

No que toca à concretização do algoritmo, é implementado um ciclo para cada nível de profundidade, sendo que para cada uma das iterações desse ciclo é realizada uma pesquisa em profundidade semelhante à discutida anteriormente.

4) Custo uniforme (Uniform cost): A pesquisa de custo uniforme assemelha-se à pesquisa em largura, distinguindo-se na escolha do nó a expandir num mesmo nível de profundidade; ao invés da pesquisa em largura, na qual esta escolha é realizada de forma aleatória, consoante a ordem de inserção dos nós na estrutura de dados utilizada, na pesquisa de custo uniforme é dada prioridade ao nós cujo custo de fronteira é menor. Quando este custo é semelhante para todos os nós, este tipo de pesquisa é igual à pesquisa em largura. Trata-se de um método completo e ótimo.

Como o custo de fronteira para nós do mesmo nível é sempre igual, visto que cada jogada tem obrigatoriamente um custo unitário, o algoritmo é essencialmente uma pesquisa em largura; contudo, tendo em conta que esta implementação utiliza uma *PriorityQueue* e que a pesquisa em largura utiliza uma *ArrayList*, ocorrem diferenças de desempenho. Isto porque a inserção de elementos de prioridade semelhante na *PriorityQueue* é aleatória, pelo que pode diferir da ordenação da *ArrayList* presente na pesquisa em largura. Isto reflete-se em desempenhos díspares, mesmo que, teoricamente e nestas condições em particular, os dois métodos sejam iguais.

C. Pesquisa informada

Os algoritmos de pesquisa informada utilizam a informação disponibilizada no problema em questão de forma a proporcionar uma procura de soluções mais eficiente. Para isto, é definida a ordenação mais apropriada dos nós a explorar, processo este que varia significativamente consoante a heurística usada.

No presente trabalho encontram-se implementadas 3 heurísticas distintas, representadas pelas classes mencionadas anteriormente:

- ***BlockNumHeuristic***: O valor da heurística é calculado segundo o número de blocos restantes no tabuleiro.
- ***ColorHeuristic***: O valor da heurística é calculado a partir do número de cores e peças restantes no tabuleiro.
- ***CloseChainHeuristic***: O valor da heurística é calculado a partir do número de cadeias que são passíveis de ser formadas através de apenas uma jogada e do número de blocos.

Apenas a heuristica CloseChain é otimista, pelo que as restantes podem não resultar em soluções ótimas.

1) Pesquisa gulosa (Greedy search): Na pesquisa gulosa, é explorado o nó que parece estar mais próximo da solução. Esta análise é realizada com recurso a uma heurística apropriada, que possa refletir com algum grau de certeza uma aproximação ao objetivo. Enquanto que a complexidade espacial é sempre exponencial, a complexidade temporal pode ser consideravelmente reduzida consoante a qualidade da heurística escolhida. Trata-se de um algoritmo não completo e não ótimo.

Enquanto que na pesquisa uniforme é utilizado o valor do custo total do custo de fronteira para chegar do nó raíz ao nó a ser avaliado para determinar a sua ordem na *PriorityQueue*, na pesquisa gulosa é usado o valor da heurística escolhida.

2) Pesquisa A (A star):* O algoritmo A* agrega as propriedadas das pesquisas de custo uniforme e gulosa, procurando minimizar a soma do caminho já efetuado com o mínimo previsto que falta até a solução. Tanto a complexidade temporal como a espacial são exponenciais. Trata-se de um algoritmo completo e ótimo caso a heurística usada seja admissível (não sobrestime a distância à solução).

Assim, a ordem de um nó na *PriorityQueue* é determinada tanto pelo custo de fronteira bem como pela heurística.

VII. Experiências e resultados

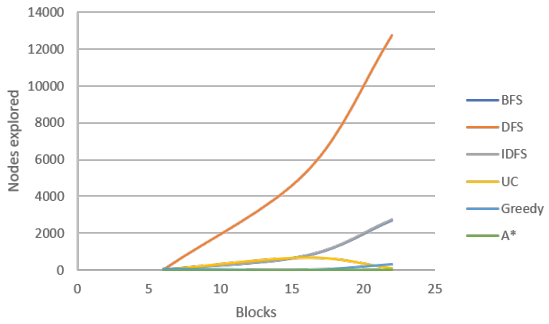

Figura 4. Desempenho espacial dos algoritmos de pesquisa em função do número de blocos de um nível

No gráfico anterior pode-se constatar o número de nodes visitados por cada algoritmo mediante o número de blocos presentes no nível o que nos permite concluir a eficiência espacial de cada um dos métodos de pesquisa.

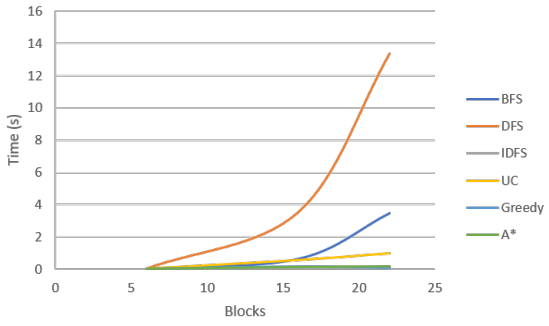

Figura 5. Desempenho temporal dos algoritmos de pesquisa em função do número de blocos de um nível

No gráfico anterior pode-se observar o tempo de procura de cada uns dos algoritmos de pesquisa em função do número de blocos de um nível. Nos gráficos seguintes é apresentado o gráfico de cada algoritmo numa forma mais específica, sendo que as diferentes curvas representam resoluções paralelas com um número diferente de movimentos (ou seja, a profundidade da solução será igual ao número de movimentos).

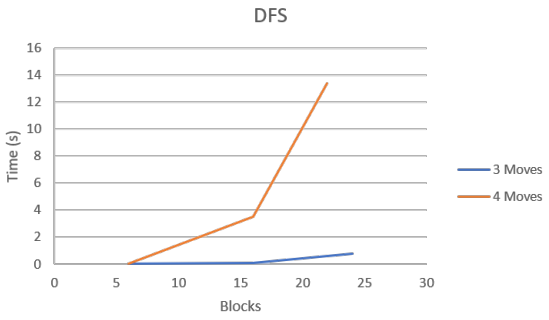

Figura 6. Desempenho temporal da pesquisa em profundidade em função do número de blocos de um nível

Observando os dados presentes no gráfico anterior observa-se o esperado, um aumento do tempo com o aumento da profundidade da árvore (mais nodes para analisar) e um aumento linear com a presença de mais blocos. O crescimento repentino no ponto do gráfico com 16 blocos deve-se ao facto da complexidade do algoritmo ser linear e terem sido usados 3 níveis para formulação destes gráficos.

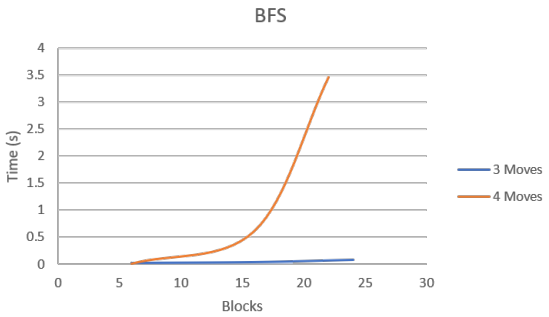

Figura 7. Desempenho temporal da pesquisa em largura em função do número de blocos de um nível

A análise deste gráfico possibilita a visualização do crescimento hiperbólico do tempo com o aumento do número de blocos, isto porque cada nível terá n vezes o número de nodes do nível anterior, sendo que n cresce com o número de blocos existentes (já que o número de operações é constante). Sendo 6 o número de operadores, b o número de blocos do node e n o nível em questão, assumindo que todos os nodes têm o mesmo número de blocos (o que teoricamente não é correto, servindo apenas para efeitos de demonstração) conclui-se que: Nodes(n) = (6*b)ˆn

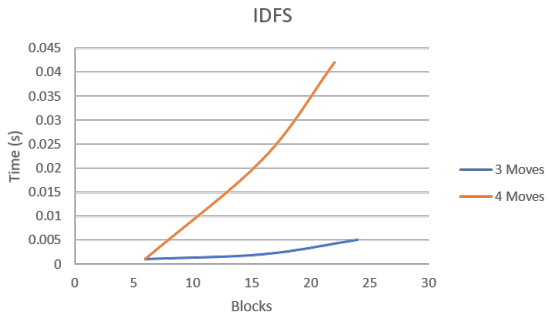

Figura 8. Desempenho temporal da pesquisa em aprofundamento iterativo em função do número de blocos de um nível

Perante este gráfico, observa-se, como no BFS, o crescimento hiperbólico do tempo com o aumento do número de blocos. O aumento de eficiência deste algoritmo em comparação à pesquisa em largura justifica-se pelo facto de que, a cada iteração, os nós são visitados da mesma maneira que seriam numa pesquisa em profundidade, não como numa pesquisa por largura.

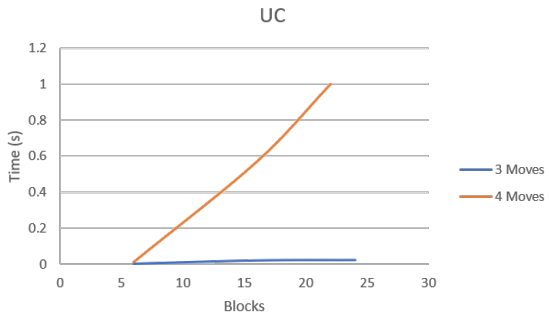

Figura 9. Desempenho temporal da pesquisa de custo uniforme em função do número de blocos de um nível

Após análise do gráfico, observa-se um aumento do tempo com o aumento da profundidade da árvore (mais nodes para analisar) e um aumento linear com a presença de mas blocos. Devido ao facto de todas arestas terem valor 1, o peso de cada nó será a profundidade do mesmo, o que faz com que o algoritmo de pesquisa de custo uniforme pesquise os nodes nível a nível à semelhança do BFS, mas é de notar que, devido aos nós no BFS serem guardados numa *ArrayList* e no UC numa *PriorityQueue*, a ordem dos nós será diferente, logo, a probabilidade do nó ótimo (que é único) ser encontrado no mesmo tempo de execução é baixa.

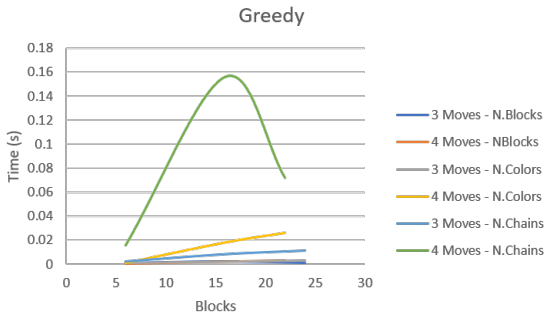

Figura 10. Desempenho temporal da pesquisa gulosa em função do número de blocos de um nível

Este gráfico , tal como o do A* que vem a seguir, contém 6 linhas em vez de 2, devido às diferentes heuristicas que foram implementadas. O elemento que salta automaticamente à vista é a linha verde. Este padrão parabólico deve-se ao facto de o Break The Ice ser um jogo extremamente difícil para o qual definir heuristicas. Por consequência, as heurísticas desenvolvidas podem ser uma ótima opção num certo tipo de nível (i.e. *ColorHeuristic* num nível em que as cores são destruídas uma de cada vez) e uma opção pouco adequada noutro tipo de nível (i.e. *BlockNumHeuristic* num nível em que os blocos são posicionados de forma a que o último move os destrua todos duma vez). Acontece então a situação representada pela linha verde no gráfico (nas outras pode acontecer também, contudo não é percetível no gráfico) em que o segundo nível usado para testar (são 3 com 6, 16 e 22 blocos) não é propício à *CloseChainHeuristic*.

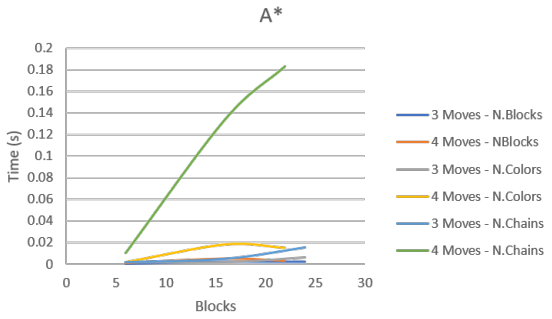

Figura 11. Desempenho temporal da pesquisa A* em função do número de blocos de um nível

Perante análise do gráfico do A*, conclui-se que a parábola presente na linha verde do algoritmo guloso desapareceu; isto deve-se à forma como os nodes são priorizados no A* em relação ao guloso. Os valores parecem estar dentro do esperado sendo uma parábola observada no traço amarelo, facilmente justificada com a dominância da heurística no cálculo do valor do node em relação ao peso e com a mesma explicação dada para o traço verde anteriormente no gráfico do algoritmo guloso.

Level	Method	Time spent (ms)	Depth	Number of nodes
1	DFS	18	2	19
	BFS	14	1	9
	IDDFS	1	1	10
	Uniform Cost	8	1	8
	Greedy-blocks	1	1	2
	Greedy-colors	1	1	2
	Greedy-chains	4	1	2
	A*-blocks	2	1	2
	A*-colors	0	1	2
	A*-chains	2	1	2
2	DFS	85	2	117
	BFS	44	1	24
	IDDFS	1	1	25
	Uniform Cost	28	1	25
	Greedy-blocks	2	1	2
	Greedy-colors	3	1	2
	Greedy-chains	5	1	2
	A*-blocks	2	1	2
	A*-colors	2	1	2
	A*-chains	8	1	2
3	DFS	849	2	419
	BFS	86	1	31
	IDDFS	2	1	32
	Uniform Cost	19	1	14
	Greedy-blocks	2	1	2
	Greedy-colors	2	1	2
	Greedy-chains	9	1	2
	A*-blocks	2	1	2
	A*-colors	5	1	3
	A*-chains	11	1	2
4	DFS	1	3	4
	BFS	6	2	34
	IDDFS	1	2	38
	Uniform Cost	5	2	28
	Greedy-blocks	1	2	3
	Greedy-colors	1	2	8
	Greedy-chains	14	4	62
	A*-blocks	1	2	3
	A*-colors	2	2	8
	A*-chains	9	2	11
5	DFS	4588	3	5311
	BFS	804	2	788
	IDDFS	16	2	806
	Uniform Cost	657	2	690
	Greedy-blocks	2	2	3
	Greedy-colors	21	4	734
	Greedy-chains	108	3	42
	A*-blocks	2	2	3
	A*-colors	11	2	10
	A*-chains	112	2	16
6	DFS	16064	3	12755
	BFS	3770	2	2705
	IDDFS	57	2	2746
	Uniform Cost	143	2	2746
	Greedy-blocks	2	2	3
	Greedy-colors	28	4	642
	Greedy-chains	66	4	297
	A*-blocks	4	2	3
	A*-colors	12	2	10
	A*-chains	174	2	21
7	DFS	200	4	704
	BFS	109	3	394
	IDDFS	9	3	422
	Uniform Cost	320	3	1076
	Greedy-blocks	6159	4	117367
	Greedy-colors	7432	4	117360
	Greedy-chains	13	3	15
	A*-blocks	609	4	662
	A*-colors	51	4	154
	A*-chains	210	3	138
8	DFS	3422	5	17114
	BFS	11136	4	38949
	IDDFS	654	4	41853
	Uniform Cost	21750	4	36722
	Greedy-blocks	14	6	936
	Greedy-colors	3	6	73
	Greedy-chains	499	6	2019
	A*-blocks	67	6	408
	A*-colors	64	6	744
	A*-chains	1780	4	772

Figura 12. Valores obtidos nos testes dos vários níveis implementados **Gold** = optimal solution **Silver** = 1 extra move **Bronze** = 2 extra moves

VIII. Conclusões e perspetivas de desenvolvimento

Em geral, consideramos que o trabalho foi realizado com sucesso, sendo que os aspetos mais relavantes para a unidade curricular, nomeadamente a implementação dos algoritmos de pesquisa, foi concretizada adequadamente. Assim, da conclusão deste projeto resultou a implementacao, na linguagem Java, de uma versão simplificada do jogo Break the Ice, no qual o utilizador pode escolher resolver diversos níveis com recurso aos algoritmos de pesquisa anteriormente mencionados.

As dificuldades mais relevantes encontradas no desenvolvimento do presente trabalho relacionaram-se com a implementação de heurísticas eficazes e otimistas, bem como nas discrepâncias entre o desempenho da pesquisa uniforme e da pesquisa em largura. Dada a natureza do problema em mãos, a conceção de uma heurística admissível mostrou ser um dos maiores desafios, uma vez que nao existem critérios claros relativamente ao caminho mais promissor para uma solução. Como tal, apenas uma das três heurísticas implementadas é otimista.

Dada a importância atribuída à concretização dos algoritmos de pesquisa, alguns aspetos secundários poderiam ser melhorados, tal como um aumento de níveis disponiveis de origem, a implementação de uma interface gráfica e a modificação da implementação da *PriorityQueue* de forma a garantir de forma a comportar-se como uma *queue FIFO* caso os elementos possuam a mesma prioridade.

Em conclusão, a realização deste trabalho contribuiu para uma consolidação dos conhecimentos adquiridos no decorrer das aulas da unidade curricular de IART, pelo que julgamos ter sido uma mais valia para o percurso académico dos membros do grupos.

Referências

[1] https://play.google.com/store/apps/details?id=com.bitmango.breaktheicehl=en
[2] https://github.com/dkarageo/ThmmyCrushCompetition
[3] http://www.cs.huji.ac.il/ ai/projects/2014/crushingCandyCrush/report.pdf