Break the Ice – Snow World (Grupo 34)

Francisco Ferreira (201605660)

MIEIC  
Faculdade de Engenharia da Universidade do PortoPorto, Portugal  
201605660@fe.up.pt

Francisco Friande (201508213)

MIEIC  
Faculdade de Engenharia da Universidade do PortoPorto, Portugal  
201508213@fe.up.pt

João Pedro Fidalgo (201605237)

MIEIC  
Faculdade de Engenharia da Universidade do PortoPorto, Portugal  
201605237@fe.up.pt

Pretende-se neste trabalho implementar um jogo do tipo solitário para um jogador e resolver diferentes versões/quadros desse jogo, utilizando métodos de pesquisa adequados. Os métodos aplicados devem ser comparados a diversos níveis com ênfase para a qualidade da solução obtida, número de operações executadas e tempo despendido para obter a solução

Keywords— pesquisa em largura, pesquisa em profundidade, pesquisa gulosa e algoritmo A\*.

# Introdução

Este trabalho foi desenvolvido no âmbito da unidade curricular de Inteligência Artificial, do 3º ano do Mestrado Integrado em Engenharia Informática e Computação, cujo objetivo é aprofundar os conhecimentos previamente adquiridos tanto nas aulas teóricas, juntamente com a abordagem de problemas práticos, sendo que o grupo optou por utilizar a linguagem C++. Sendo assim, o grupo propôs-se a implementar o jogo de tabuleiro Break the Ice com os seguintes modos: resolução do computador e resolução do jogador, sendo que este pode pedir dicas sobre qual a melhor jogada possível.

O relatório está dividido nas seguintes secções: Introdução; Descrição do Problema; Formulação do Problema; Trabalho Relacionado; Implementação do Jogo; Algoritmos de Pesquisa; Experiências e Resultados; Conclusões e Perspetivas de Desenvolvimento; Referências Bibliográficas.

# Descrição do Problema

O jogo Break the Ice caracteriza-se por ser um jogo de tabuleiro do tipo solitário cujo objetivo é fazer com que todas as peças em jogo sejam eliminadas, resultando numa matriz vazia. Para que as peças possam ser eliminadas é necessário que estas formem uma linha(horizontal ou vertical) com um número de peças igual ou superiores a três com a restrição de que todas as peças dessa linha têm obrigatoriamente de possuir a mesma cor.

Para que uma peça seja movimentada tem que haver uma peça adjacente com diferente cor e a movimentação consiste na troca das suas posições.

Esta troca de posições tem repercussões, nomeadamente a formação de um grupo de três ou mais peças da mesma cor, eliminando por isso o grupo referido. Uma peça não pode estar em cima de um bloco vazio, ou seja, é necessário que esteja diretamente em contacto com o nível do solo (ultima linha da matriz) ou então que esteja a ser suportada por uma peça abaixo de si. Assim sendo, se uma peça que esteja suportar outra for eliminada, a segunda vai descer no tabuleiro até encontrar uma outra peça ou a base do tabuleiro.

# Formulação do Problema

1. Representação do estado

O estado do jogo vai ser representado por uma matriz de duas dimensões em que cada elemento representa um bloco de jogo, este pode estar vazio ou com uma peça/cubo. Cada peça pode ter 6 cores possíveis (roxo, laranja, rosa, azul, verde, amarelo).

Estado Inicial:

As peças estarão distribuídas pelo fundo da matriz e, inicialmente, não há nenhum conjunto de 3 cubos alinhados, nem com espaços por baixo, visto que estes caem.

Teste objetivo:

Não haver nenhuma peça em jogo, ou seja, a matriz estar com todos os elementos vazios.

Operadores:

- Mover Peças/Cubos horizontalmente

Pré-condições - O elemento selecionado ser uma peça/cubo e este não pode ser movido contra os limites da matriz.,

Efeitos - A peça movimenta-se horizontalmente, conforme o selecionado, para a “casa” ao lado e caso a nova posição esteja sem peças por baixo, esta irá cair até encontrar o fundo ou outra peça.

Custo: 1

.

- Trocar Peças/Cubos horizontal e verticalmente:

Pré-condições - As peças têm de estar em posições adjacentes (horizontal ou verticalmente).

Efeitos - As peças trocam de posição uma com a outra.

Custo: 1

Efeitos gerais:

Após a movimentação das peças, se houver 3 ou mais peças com a mesma cor em linha, na vertical ou horizontal, essas irão desaparecer e todas as que estão nas posições superiores a elas “cair”.

Custo da solução:

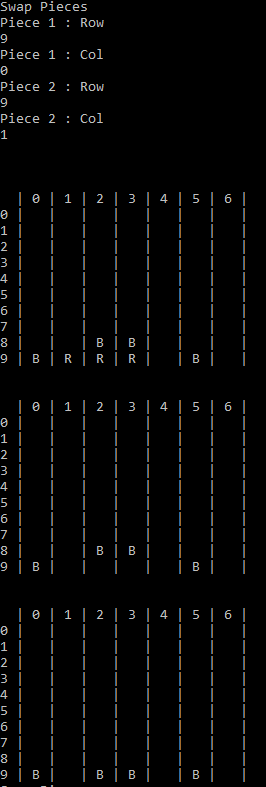
Cada movimento custa 1, sendo o custo da solução o número de movimentos para resolver o puzzle.

# Trabalho Relacionado

O trabalho relacionado diz respeito à implementação do jogo/puzzle e aos métodos de pesquisa envolvidos na sua resolução. As hiperligações incluídas nesta secção incluem no seu conteúdo código fonte referente ao respetivo mecanismo a ser usado em C++., relativamente à criação do tabuleiro e jogabilidade[1]; e Métodos de Pesquisa usados[2] [3][4].

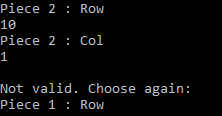
# Implementação do Jogo

O jogo foi implementado em C++, sendo o display do estado do tabuleiro feito através do terminal, tal como toda a jogabilidade e seleções de menu, através de impressões e leituras. Cada nível tem um número perfeito de jogadas e um máximo, que é superior em 2 ao número perfeito. O display do tabuleiro é feito sempre que há uma alteração no mesmo, nisto inclui-se mover uma peça, eliminar peças ou fazer cair peças, ou seja, para um operador de movimentação pode haver vários displays seguidos do tabuleiro.

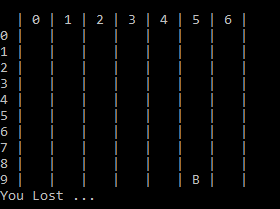


Algumas das restrições do jogo são:

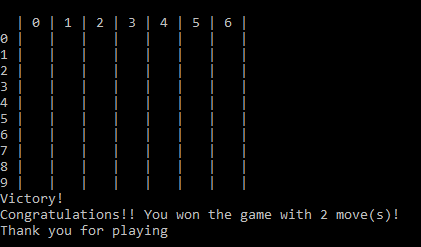
- O jogador/computador não conseguem mover peças para um espaço vazio na vertical, nem fazer qualquer tipo de movimentação para fora do tabuleiro.



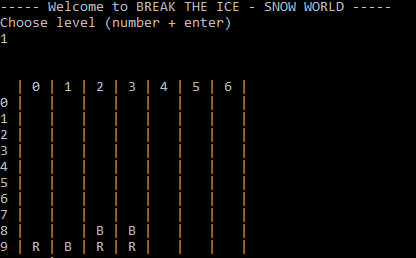
- Quando o número de peças de uma certa cor está entre ]0,3[, o jogo acaba com uma derrota pois já não é possível eliminar essas mesmas peças.



- Quando já não há peças existentes no tabuleiro, o jogo termina com uma mensagem de vitória.



Os vários níveis estão a ser lidos de ficheiros, quando se entra no jogo é dada a opção de escolher que nível se quer jogar, bastando introduzir o número desse mesmo nível como input.



# Algoritmos de Pesquisa

Para a solução do problema, o mesmo foi formalizado em forma de árvore, correspondendo o estado de jogo resultante de uma jogada a cada um dos nós, considerando todas as jogadas possíveis de todas as peças. Os níveis da árvore correspondem ao número de jogadas.

De seguida apresenta-se a estrutura de dados de um nó e as estruturas utilizadas para os algoritmos.

struct node{

vector<int> from;

vector<int> to;

int level;

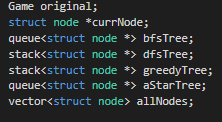
float value;

float heuris;

Game prevGame;

struct node \*father;

};



Foram usados para a determinação das soluções dos níveis os seguintes algoritmos de pesquisa: em largura, em profundidade, A\* e gananciosa (ou gulosa). [5][6] Em todos a árvore apenas atinge um nível igual ou inferior ao número máximo de jogadas. Todos os nós são guardados num vetor para se conseguir verificar se estes são duplicados. Os estados em que o tabuleiro é duplicado e o nó a ser analisado tem um nível superior ao já existente este deixa de ser desenvolvido.

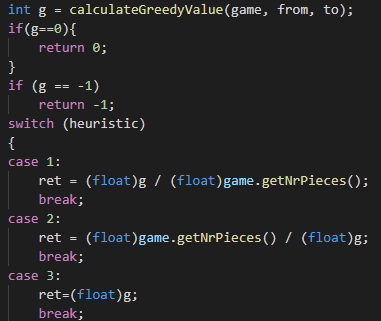
Em todos os algoritmos o modo de funcionamento tem como base:

Um loop para verificar todas as movimentações aceites para cada peça do tabuleiro atual. Adicionar as movimentações à estrutura de dados utilizada para esse algoritmo. Para o caso do algoritmo de pesquisa gulosa e A\* estas movimentações são adicionadas tendo em conta o seu valor. Depois escolher o próximo nó, dependendo do algoritmo, para ser desenvolvido. Enquanto não se encontrar uma solução ele continua a correr. Não são considerados os nós que atingem um estado de derrota. Se a estrutura de dados desse algoritmo atingir um estado em que o seu tamanho é 0, retirando a primeira vez que os loop corre, quer dizer que o algoritmo não encontrou uma solução e por isso há algum tipo de erro no nível definido.

O algoritmo de pesquisa em largura começa por desenvolver todos os nós sucessores do *root* node, depois todos os filhos destes e assim sucessivamente para todos os restantes. Há, então, uma expansão de todos os nós (passíveis de expansão) de uma dada profundidade antes de qualquer nó do nível seguinte. Este utiliza como estrutura de dados uma queue chamada de bfsTree.

O algoritmo de pesquisa em profundidade desenvolve sempre o nó a ser expandido até a sua profundidade máxima e, quando este não tem mais possibilidades, passa para o seguinte que encontra mais fundo e assim sucessivamente para todos os restantes. Este utiliza como estrutura de dados uma stack chamada de dfsTree.

A\* é um algoritmo de pesquisa em que se combina o custo definido para operações com uma função de heurística apropriada para a chegada à solução ser mais eficiente, dado que a procura da mesma terá uma estratégia de decisão. Este utiliza como estrutura de dados uma queue chamada de aStarTree. Foram usadas 3 heurísticas diferentes.

A primeira consiste na divisão do número de peças do seu tabuleiro pelo número de peças do tabuleiro do pai.

A segunda no inverso da primeira.

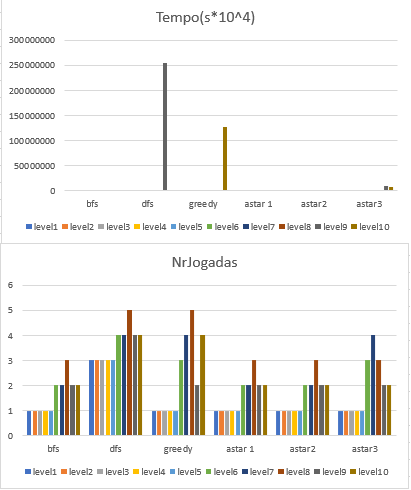
A terceira utiliza o mesmo que o greedy, ou seja, o número de peças no seu tabuleiro.

Pesquisa gulosa, ou gananciosa apresenta uma estratégia de decisão pouco eficiente, escolhendo sempre a melhor opção que encontrar localmente, sem ter o panorama geral em consideração. Optamos por definir que um tabuleiro é melhor que outro caso tenha menos peças. Este utiliza como estrutura de dados uma stack chamada de greedyTree.

# Experiências e Resultados

Foram executados vários testes de eficiência dos algoritmos de forma a comparar os custos da solução em cada um e os resultados a nível temporal. É de notar que todos os tempos apresentados incluem a impressão dos tabuleiros, para efeitos demonstrativos, de modo a que todos os valores sejam comparados já com a agravante referida.

Os testes foram de tempo de execução e de número de jogadas por cada nível para cada algoritmo.



Para cada nível o número máximo de jogadas é superior em 2 em relação ao número perfeito de jogadas. O número perfeito de jogadas para cada nível é o seguinte:

Nível 1- 1

Nível 2- 1

Nível 3- 1

Nível 4- 1

Nível 5- 1

Nível 6- 2

Nível 7- 2

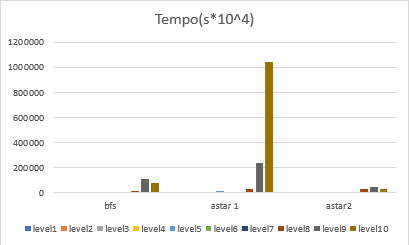
Nível 8- 3

Nível 9- 2

Nível 10- 2

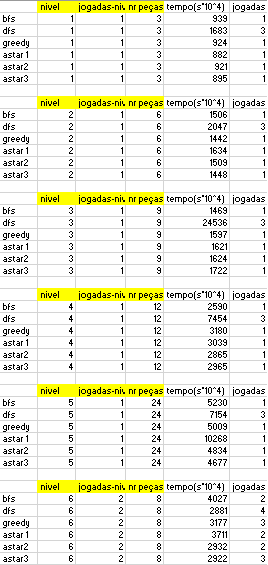
Como se pode verificar em questão de tempo de execução verifica-se que a pesquisa em profundidade e gulosa tem tempos de execução, em certos níveis, tão superiores aos restantes algoritmos que não se consegue perceber os tempos dos restantes.

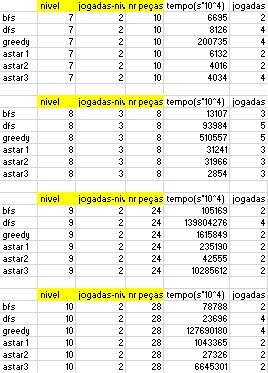
Em questão de número de jogadas verifica-se que oes melhores são a pesquisa em largura, A\* com a primeira e segunda heurística, ambas com um número perfeito de jogadas. Para melhor entendimento do melhor algoritmo, como ambos, nos testes realizados, dão o mesmo número de jogadas, comparou-se o tempo de execução destes três algoritmos.



Após esta comparação verifica-se que o que tem tempos de execução menores é o A\* utilizando a segunda heurística.

Foram registados também todos os dados relativos ao número de peças e número de jogadas para cada nível.





Analisando estes registos verifica-se que quanto maior o número de peças, maior o tempo de execução de cada algoritmo, tal como com o aumento do número de jogadas para cada nível.

# Conclusões e Perspetivas de Desenvolvimento

Com este trabalho conseguimos aprender um pouco mais sobre a inteligência artificial, com a criação de árvores para se chegar à solução, com a implementação de diferentes algoritmos de pesquisa dessa mesma solução e com a execução de testes para verificação de qual o mais eficiente.

No caso do trabalho desenvolvido por nós, o algoritmo que melhor se encaixaria numa implementação de inteligência artificial seria o algoritmo de pesquisa em largura, dado que os testes revelaram uma grande eficiência, tanto a nível de solução, como a nível temporal, mesmo para casos que obrigavam a muitas jogadas para o estado de aceitação.

##### Referências Bibliográficas

1. [https://cboard.cprogramming.com/c-programming/166156-creating-match-3-candy-crush-game-using-arrays.html](https://cboard.cprogramming.com/c-programming/166156-creating-match-3-candy-crush-game-using-arrays.html?fbclid=IwAR1xhwyvpYH7AiIrxcpv3R4Ai1I7FYWCIxP_p1Tl6NysLaP8j567C4jg0J4)
2. [https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/](https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/?fbclid=IwAR1X8EQUu5_K_5Z8E_gbD4QJO49FaPjL6NQ-qUofAr9tVB8bJoY9Cy-QnGM)
3. [https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/](https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/?fbclid=IwAR3ZYEkBpnx1v-5LCUxiGda8giupPPHSGQ3ArqpwhTuT_ahhiA1OJorq3vU)
4. [https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/](https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/?fbclid=IwAR1rKLXtcpdulG_60axjFHRuP61Ih7aft0KmmTrrnLDMVgrLvERxAmzTR_M)
5. Stuart Russel and Peter Norvig, “Artificial Intelligence: A Modern Approach”, Third Edtition, Pearson Education Inc., 2010, ISBN: 978-0-13-604259-4.
6. Stuart Russel and Peter Norvig, “AimaCode - Code for the Book Artificial Intelligence: A Modern Approach", 2019, [online], available at: <https://github.com/aimacode> , consulted on March 2019.