Resolução do Cohesion utilizando Métodos de Pesquisa em Linguagem Python (Tema 4/Grupo 7)

João Miguel Vaz Tello da Gama Amaral (up201708805)
Departamento de Engenharia Informática
Faculdade de Engenharia da Universidade do Porto
Porto, Portugal
up201708805@fe.up.pt

João Nuno Rodrigues Ferreira (201605330)
Departamento de Engenharia Informática
Faculdade de Engenharia da Universidade do Porto
Porto, Portugal
up201605330@fe.up.pt

Tiago Pinho Cardoso (201605762)
Departamento de Engenharia Informática
Faculdade de Engenharia da Universidade do Porto
Porto, Portugal
201605762@fe.up.pt

***Resumo*— Pretende-se neste trabalho implementar um jogo do tipo solitário para um jogador e resolver diferentes versões desse jogo, utilizando métodos de pesquisa. Para esta entrega intercalar será descrito neste artigo o problema, a formulação do mesmo, trabalho relacionado e conclusões e perspetivas de desenvolvimento.**

Keywords — Inteligência Artificial, Pesquisa, Pesquisa em Largura, Pesquisa em Profundidade, Pesquisa de Aprofundamento Progressivo, Pesquisa de Custo Uniforme, Pesquisa Gulosa, Algoritmo A*

I. Introdução

Este trabalho consiste na implementação de Métodos de Pesquisa no âmbito de resolver diferentes versões do 7 jogo escolhido, do tipo solitário. Para esse efeito, será necessário utilizar os métodos adequados, sejam pesquisa em largura, pesquisa em profundidade, aprofundamento progressivo, pesquisa de custo uniforme, pesquisa gulosa e algoritmo A*.

Um jogo do tipo solitário caracteriza-se pelo tipo de tabuleiro, peças, regras de movimentação, condições de terminação do jogo com derrota ou vitória e com a respetiva pontuação.

Pretende-se desenvolver então uma aplicação para jogar estes jogos, neste caso, Cohesion. A aplicação deverá ter uma visualização em modo de texto ou gráfico para mostrar a evolução do tabuleiro e realizar a comunicação com o utilizador/jogador.

Para esta entrega final serão incluídos no artigo a descrição do problema, formulação do problema, trabalho relacionado, implementação do jogo, algoritmos de pesquisa, experiências e resultados e conclusões e perspetivas de desenvolvimento. A descrição do problema consiste numa sucinta descrição do solitário e as suas regras. A formulação do problema irá descrevê-lo como um problema de pesquisa, descrevendo a representação do estado, teste objetivo, operadores, pré-condições, efeito e custo. A implementação do problema irá descrever o jogo, bem como a sua implementação na linguagem Python. Nos algoritmos de pesquisa, estes serão descritos tal como, uma vez mais, a sua implementação. Por fim, as experiências e resultados irão revelar os resultados dos vários algoritmos com diversos puzzles.

II. Descrição do Problema

Cohesion é um puzzle baseado no clássico "15 puzzle", também conhecido por "slide the square" ou "slide puzzle", porém o Cohesion difere de uma maneira importante.

Os quadrados do jogo Cohesion têm várias cores. Se dois quadrados da mesma cor se tocam, eles unem-se formando uma só peça. A ligação é permanente e a nova forma criada aumenta o desafio de Cohesion.

O puzzle é resolvido assim que todos os quadrados de cada cor forem combinados em uma só peça. O jogo está organizado em 240 puzzles categorizados em três níveis de dificuldade. Na imagem seguinte podemos ver uma possível resolução de um dos níveis.

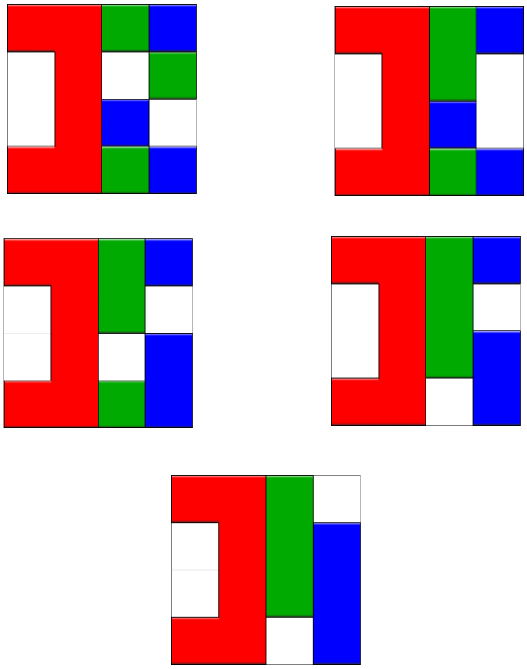

Fig. 1 – Possível resolução de um nível. A ordem das imagens é da esquerda para a direita e de cima para baixo.

III. Formulação do Problema

Representação do estado do jogo: O jogo irá ser representado por uma matriz de tamanho 4x4. Nesta matriz, todos os elementos irão ser representados por um valor que se encontra no intervalo entre 0 e N, sendo que 0 representa uma posição sem qualquer elemento e todos os outros valores até N representam uma cor diferente das outras.

Estado Inicial: O estado inicial irá depender do nível escolhido. A única restrição neste campo, é que todas as peças da mesma cor não podem estar juntas visto que isso iria corresponder a um estado final.

Operadores: Há quatro ações possíveis: esquerda, direita, cima ou baixo. Cada uma destas ações ocorre numa peça ou num conjunto de peças adjacentes da mesma cor. Para que a ação possa ser realizada, tanto a peça como o conjunto não

podem estar a ser bloqueadas por outras peças, ou seja, só se pode mover um bloco se todas as peças que constituem este apenas tiverem uma peça da mesma cor ou um espaço vazio na direção desejada.

Teste objetivo: O teste objetivo irá verificar se o estado atual do jogo é um estado final, sendo que este é um estado no qual todas as peças da mesma cor se encontram juntas.

Custo da solução: Neste caso, o custo de cada passo será um. Assim sendo, o custo da solução que resolve o problema é o número de passos para chegar à tal solução.

IV. Trabalho RelacionadoS

No caso do jogo escolhido pelo nosso grupo, Cohesion, cuja página no Google Play é [1], não nos foi possível encontrar código fonte do jogo em si.

Por outro lado, encontrámos implementações de algoritmos que nos serão úteis para a realização do projeto no site [2], visto que a implementação destes é já feita em Python, linguagem na qual iremos trabalhar.

V. Implementação do Jogo

Este projeto foi realizado na linguagem Python. Para implementar o jogo foi necessário estabelecer uma forma de representação do estado do jogo, bem como operadores que afetam o estado do jogo ou que o avaliam e funções auxiliares como a que lê os níveis de um ficheiro de texto.

Inicialmente, são chamadas as funções *parseFile*, *chooseAlg* e *chooseLevel* que, respetivamente, leem de um ficheiro de texto todos os níveis possíveis e fornecem ao utilizador a possibilidade de escolher o nível e o algoritmo que o irá resolver. De seguida, é criada uma instância da classe Game, onde se começa por criar um tabuleiro - variável *board* (array de objetos da classe Piece) - com a função *createPieceBoard* e um array de objetos da class Block – variável *blocks* (que é um conjunto de Pieces que se encontrem juntas) - com a função *createBlocks*.

Tendo por base as variáveis referidas no parágrafo anterior, são inicializados os métodos de pesquisa que serão descritos na secção seguinte.

Por fim, é necessário também referir a implementação das funções que são essenciais para a realização dos movimentos que são elas *ableToMove*, *moveBlock*, *clean_blocks*, para listar todas as jogadas disponíveis, *gameChilds*, e para testar se um estado é solução, *checkWin*. A primeira função recebe como parâmetros um game, um bloco e uma direção e retorna 0 ou 1, dependendo se o movimento é possível ou não. O predicado *moveBlock* é chamado com parâmetros iguais ao anterior, caso seja exequível a realização do movimento. Já a *clean_blocks* é chamada após a *moveBlocks* e realiza a "limpeza" dos blocos, ou seja, peças da mesma cor que se encontrem juntas, mas que só após a última jogada, passam a fazer parte do mesmo block. Em relação à *gameChilds*, esta recebe um estado e retorna todos os estados filho, sendo estes aqueles que sucedem ao estado recebido como parâmetro. Ultimamente, a função *checkWin* retorna se um estado é solução, fazendo a verificação se apenas existe, para cada cor, um bloco.

VI. Algoritmos de Pesquisa

Neste projeto foram implementados e utilizados vários algoritmos de pesquisa, sendo estes, pesquisa em largura, em profundidade, de aprofundamento progressivo, de custo uniforme, gulosa e algoritmo A*. Todos têm como objetivo obter um estado solução partindo de um estado inicial.

O algoritmo de pesquisa em largura, função *breadthFirst* no ficheiro "*tree.py*", recebe como argumento o estado inicial do jogo. Primeiramente, é inicializado um array que apenas possui o estado inicial e, de seguida, é executado um ciclo no qual se percorrem os estados filho do estado que está a ser verificado, ou seja, todos os estados que se encontram a apenas um movimento de distância do pai. Percorrendo os filhos, caso este seja um estado final, já foi encontrada a solução, caso contrário, se este não for um estado que tenha sido verificado, é colocado no array anteriormente falado de forma a ser analisado no futuro.

Já o algoritmo de pesquisa em profundidade, função *depthFirst*, é bastante semelhante ao método anterior, com a ligeira diferença de que, em vez de um array, é usada uma stack para guardar todos as instâncias de game que ainda terão de ser observadas. Desta forma, dá-se prioridade aos filhos de um estado em vez de aos nós que se encontram no mesmo nível deste.

O método de aprofundamento progressivo, função *progressiveDeepening*, pode ser considerado uma mistura dos dois últimos. Este, para além do estado inicial, também recebe valor que representa o quão progressivo o algoritmo será. Tal como na pesquisa em profundidade, é usada uma stack para guardar os estados ainda não verificados, mas é também utilizado um array, variável *toBeChecked*, que será explicado ainda neste parágrafo. Inicialmente, o algoritmo porta-se exatamente como o de pesquisa em profundidade, mas apenas até ao nível na árvore igual ao valor recebido por parâmetro. Sempre que for encontrado um estado no nível referido que ainda não é solução, este é colocado em *toBeChecked* e não se aprofunda mais esse ramo. De seguida, passa-se a fazer uma pesquisa semelhante à de largura. Quando se acabar de fazer a verificação de todos os estados abaixo do nível, não se chegando à solução, o nível de progressão é aumentado e recomeça-se a processar os estados que estavam no nível anterior.

O algoritmo de pesquisa de custo uniforme, função *uniform_cost_search*, recebe um estado inicial como argumento, "initState". São declarados dois arrays, "front" onde cada elemento corresponde a um valor calculado pela heurística e a um estado, e "expanded" que guarda os estados que já foram expandidos. De seguida é declarado um While onde são comparados dois estados do jogo (dois seguidos no array front) de acordo com o valor do custo (aumenta sempre em 1). A variável path e front são então atualizadas de acordo com o estado que possui o menor custo. É criada uma variável "endnode" que é igualada ao último elemento de path, se este for um estado final é dado append do "endnode" no path e o ciclo termina. Se "endnode" não for um estado final são calculados todos os estados atingíveis a partir de "endnode" e colocados em "newpath". Após isto é colocado em "front" o "newpath" e em "expanded" o "endnode".

O método de pesquisa gulosa, função *greedy*, recebe como argumentos uma lista "visited" e um estado inicial "initState". É declarado um array "states" que é inicializado com "initState", um valor padrão para a heurística e um "nextState" que vai representar o próximo estado a ser analisado, ou seja, o melhor filho do estado atual. De seguida, é inicializado um ciclo While enquanto o iterador "i" for menor que o

comprimento de “states”. Dentro do While são calculados os próximos estados possíveis e guardados numa variável para serem calculadas os valores da heurística em cada um deles e atualizar a melhor hipótese e a variável “heuristicValue”. Em suma, neste algoritmo, é apenas considerado um próximo estado, sendo este o que possui o melhor valor depois de aplicada a heurística. Como se trata de um algoritmo não ótimo, é normal, por vezes, não encontrar uma solução e entrar num loop. Nestes casos é imprimida a mensagem: “Couldn’t find solution”.

Por fim, o algoritmo A*, função *a_star*, recebe um estado inicial como argumento, “initState”. São declarados dois arrays, “front” onde cada elemento corresponde a um valor calculado pela heurística e a um estado, e “expanded” que guarda os estados que já foram expandidos. De seguida é declarado um While onde são comparados dois estados do jogo (dois seguidos no array front) de acordo com o valor da heurística. A variável path e front são então atualizadas de acordo com o estado que possui melhor heurística. É criada uma variável “endnode” que é igualada ao último elemento de path, se este for um estado final é dado append do “endnode” no path e o ciclo termina. Se “endnode” não for um estado final são calculados todos os estados atingíveis a partir de “endnode” e colocados em “newpath”. Após isto é colocado em “front” o “newpath” e em “expanded” o “endnode”.

Todos os algoritmos verificam uma condição no início da função, verificar se o estado inicial recebido é um estado final. Caso a condição seja verdadeira as funções retornam imediatamente.

VII. Experiências e Resultados

Aqui irão ser colocadas tabelas comparativas de resultados obtidos. Os resultados analisados e registados são relativos ao tempo e custo da solução obtida em cada nível, de cada algoritmo utilizado em diferentes puzzles.

BFS:

Level	1	2	3	14	18
Moves	4	4	4	9	8
Time (s)	2.003	0.106	0.514	1.508	6.536
Checked Nodes	238	35	72	318	1016

DFS:

Level	1	2	3	14	18
Moves	10	10	5	18	18
Time (s)	0.645	0.509	204.918	6.826	4.992
Checked Nodes	14	29	6507	168	995

Progressive deepening:

Level	1	2	3	14	18
Moves	5	4	9	10	8
Time (s)	15.629	6.637	3.688	4.753	9.206
Checked Nodes	482	47	93	153	491

Uniform cost search:

Level	1	2	3	14	18
Moves	4	4	4	9	8
Time (s)	12.656	0.19	2.231	3.582	21.302
Checked Nodes	1574	88	369	1007	3221

Greedy:

Level	1	2	3	14	18
Moves	4	4	7	Failure	Failure
Time (s)	0.037	0.016	0.035	Failure	Failure
Checked Nodes	4	4	9	Failure	Failure

A* algorithm:

Level	1	2	3	14	18
Moves	4	4	6	9	10
Time (s)	0.031	0.034	0.022	0.791	0.342
Checked Nodes	5	16	9	316	145

VIII. Conclusões e Perspetivas de Desenvolvimento

O facto de o tema deste trabalho reunir vários algoritmos de inteligência artificial, nomeadamente A*, torna a sua implementação e compreensão mais interessante. A aplicação da matéria aprendida nas aulas, tanto teóricas como práticas, contribuem para um melhor aproveitamento e aprofundamento do trabalho em questão.

Podemos concluir que a realização de experiências recorrendo a algoritmos diferentes nos permitiu perceber melhor o funcionamento dos algoritmos e que fatores influenciam os seus resultados. É de realçar que quantos mais blocos da mesma cor estiverem presentes no tabuleiro e quanto maior for a distância entre os mesmos, maior é o tempo que os algoritmos demoram a resolver o puzzle.

Em suma, o trabalho foi realizado com sucesso, tendo-se cumprido todos os requisitos e ultrapassado todos os obstáculos.

Referências Bibliográficas

[1] https://play.google.com/store/apps/details?id=com.NeatWits.CohesionFree.

[2] https://www.redblobgames.com/pathfinding/a-star/implementation.html