Resolução do jogo Cohesion utilizando Métodos de

Pesquisa em Linguagem Python (Tema 4/ Grupo 33)

João Alves (201605236)  
Faculdade de Engenharia da Universidade do Porto

Porto, Portugal

[up201605236@fe.up.pt](mailto:up201605236@fe.up.pt)

Amadeu Pereira (201605646)  
Faculdade de Engenharia da Universidade do Porto

Porto, Portugal

[up201605646@fe.up.pt](mailto:up201605646@fe.up.pt)

Nuno Lopes (201605337)  
Faculdade de Engenharia da Universidade do Porto

Porto, Portugal

[up201605337@fe.up.pt](mailto:up201605337@fe.up.pt)

*Resumo* - Este trabalho tem como objetivo a aplicação de vários métodos de pesquisa na resolução de vários níveis de um jogo (*cohesion*). A aplicação permite dois modos de jogo (jogador / computador) em que, para o computador, pode ser escolhido o método a ser utilizado para a resolução do puzzle.

O código fonte é escrito na linguagem python, o que ajudou no desenvolvimento do trabalho. A parte do computador (Inteligência Artificial) foi testada com todos os algoritmos de pesquisa desenvolvidos, variando a dificuldade dos níveis (*easy*-*medium*-*hard*) obtendo quase sempre uma resposta positiva. À medida que é aplicado um método é feita uma análise aos seus custos que é representada em modo texto pela aplicação. De um modo geral os resultados pretendidos aproximam-se aos resultados esperados confirmando a teoria estudada.

*Keywords: Inteligência Artificial, Pesquisa, Algoritmo A\*, Cohesion, Pathfinding, Procura em Profundidade, Procura em Largura, Custo Uniforme, Aprofundamento Iterativo, Pesquisa Gulosa*

# Introdução

TODO: Alguns parágrafos motivando e introduzindo o tema e os objetivos do documento e descrevendo a estrutura do artigo.

# Descrição do Problema

O jogo Cohesion é um puzzle baseado no jogo “15 Puzzle” onde os quadrados têm várias cores. As peças podem ser jogadas para qualquer lado desde que não haja nenhuma peça de outra cor que impeça.

Se dois quadrados/peças da mesma cor se tocam eles unem-se num só permanentemente, originando assim uma peça nova.

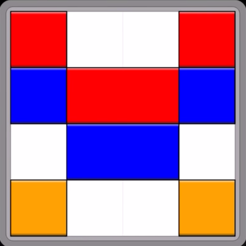


Ilustração 1 - Exemplo de um nível do puzzle.

O objetivo deste puzzle é juntar todas as peças da mesma cor.

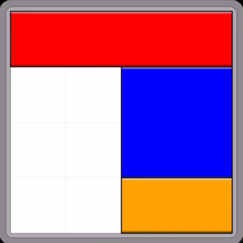


Ilustração 2 - Exemplo de um estado objetivo do puzzle.

# Formulação do Problema

Representação do Estado: Matriz 4x4 em que cada elemento pertence a {0, ..., N}, sendo N o número total de cores distintas existentes no jogo. Quando um elemento é 0 significa que não existe nenhuma peça (cor) nesta posição.

Estado Inicial: O estado inicial depende do nível em questão. Consiste numa distribuição de várias cores pela matriz, garantindo que as peças da mesma cor não se encontram todas juntas.

Estado Objetivo: As peças da mesma cor estão todas adjacentes umas às outras.

Operadores:

Quando 2 peças da mesma cor se tocam formam um bloco permanente, tendo este conjunto de peças da mesma cor de se mover em conjunto.

* **esquerda**(pos):

Pré-Condição: todas as posições à esquerda do bloco da peça selecionada(pos) tem que ser uma posição vazia.

Efeito: o bloco anda uma posição para a esquerda.

Custo: 1.

* **direita**(pos):

Pré-Condição: todas as posições à direita do bloco da peça selecionada(pos) tem que ser uma posição vazia.

Efeito: o bloco anda uma posição para a direita.

Custo: 1.

* **cima**(pos):

Pré-Condição: todas as posições em cima do bloco da peça selecionada(pos) tem que ser uma posição vazia.

Efeito: o bloco anda uma posição para cima.

Custo: 1.

* **baixo**(pos):

Pré-Condição: todas as posições em baixo do bloco da peça selecionada(pos) tem que ser uma posição vazia.

Efeito: o bloco anda uma posição para baixo.

Custo: 1.

Custo da solução: Cada movimento custa 1, logo o custo da solução é o número total de movimentos.

# Trabalho Relacionado

Não conseguimos encontrar nenhum código-fonte do jogo Cohesion, logo como tal decidimos basearmo-nos em implementações do jogo “15 Puzzle” uma vez que tem certas semelhanças ao nosso puzzle. Sendo assim encontramos a implementação de uma pessoa chamada Milan Pecov [2] onde implementa algoritmo A\* e pesquisa em largura para o jogo “15 Puzzle”.

Também para a implementação deste jogo vamos utilizar o código disponível no link [1] pois é possível encontrar a implementação do jogo em diversas linguagens de programação.

TODO: ACRESCENTAR MAIS CENAS

# Implementação do Jogo

Cohesion é um jogo para Android cujo objetivo é juntar todas as peças da mesma cor. Para a implementação deste jogo optamos por criar uma classe ***Game*** onde existe uma matriz 4x4 para representar o tabuleiro, com o estado atual do jogo, mapeando cada elemento desta matriz ao seu respetivo bloco no jogo (sendo 0 o equivalente a nenhuma peça).

Para facilitar a organização do código do jogo criamos uma classe ***Block***, que representa um bloco de peças da mesma cor, tais peças guardadas num array com as suas respetivas coordenadas (par com o x e o y). Esta classe é responsável por todas as ações envolvendo blocos, como verificar se se encontra adjacente a outro bloco em algum ponto e os operadores (***up***, ***down***, ***left***, ***right***).

def up(self):

for c in self.coords:

c[0] = c[0] – 1

def down(self):

for c in self.coords:

c[0] = c[0] + 1

def left(self):

for c in self.coords:

c[1] = c[1] – 1

def right(self):

for c in self.coords:

c[1] = c[1] + 1

Desta forma foi também necessário guardar, além do tabuleiro, todos os blocos existentes naquele jogo. Para isso, sempre que o jogo é atualizado atualiza-se os blocos existentes, através da função ***update\_blocks()*** que utiliza a função definida em ***Block*** ***check\_block\_adjacent(block)*** para determinar se dois blocos se encontram adjacentes e fazer a sua correspondente agregação.

O jogador, ao fazer um movimento este só é aceite pelo jogo se tal passar nas suas correspondentes pré-condições. Tendo como exemplo o movimento ***up*** de um bloco este só será concluído se todas as peças constituintes verificarem as seguintes condições: a posição imediatamente acima tem que estar vazia (correspondendo a um 0 na representação interna do tabuleiro) ou tem que estar ocupada por uma peça da mesma cor, uma vez que esta pertencerá ao mesmo bloco e também será movida, libertando esse espaço. Cada movimento no jogo tem um custo de 1. Se a ação for validada, o estado interno do jogo é atualizado (tanto o tabuleiro como os blocos que o constituem).

def is\_possible\_up(self, block):

for [x,y] in block.coords:

x1 = x - 1

if x1 < 0 or (self.board[x1][y] != 0 and self.board[x1][y] != block.color):

return False

return True

Para verificar se um jogo corresponde a uma condição de vitória temos de testar se todas as peças da mesma cor se encontram juntas, ou seja, se existe apenas um bloco de cada cor. Tal teste objetivo foi implementado da seguinte forma na classe ***Game***

def is\_finished(self):

colors = []

for block in self.blocks:

if block.color in colors:

return False

else:

colors.append(block.color)

return True

Além das implementações básicas para o jogo funcionar também está incorporado na aplicação a habilidade de ler os níveis de um ficheiro json (levels.json) dividido em diferentes dificuldades. Além disso também apresenta uma interface, permitindo que o utilizador visualize o jogo e faça ações de forma mais intuitiva e simples. Para o utilizador poder jogar o jogo tem que carregar em ***Play***, selecionar um bloco com o rato e escolher o movimento que pretende fazer usando as setas do seu teclado. Também está implementada uma opção de ajuda (***Hint***), que dá ao utilizador, se estiver a jogar, o melhor movimento para o jogo no seu estado atual.

Para o desenvolvimento da Inteligência Artificial tivemos que criar uma função computasse todas as jogadas possíveis num determinado estado de jogo. Esta função é ***get\_game\_moves(game)*** que recebe como parâmetro um jogo e retorna um array de objetos do tipo ***([block\_index, movement], new\_game)***. Cada objeto corresponde a um estado "filho" do dado em que ***block\_index*** corresponde ao índice do bloco movido, ***movement*** é uma string (***up***, ***down***, ***left***, ***right***) representando o movimento aplicado ao bloco, e o ***new\_game*** é o novo estado resultante. Também foi necessário implementarmos várias heurísticas. Optamos assim por desenvolver 3 delas, duas muito semelhantes entre si uma vez que ambas calculam a distância entre todas as peças da mesma cor, mas uma utiliza a distância “Euclidiana” e outra a distância de “Manhattan”. A terceira é uma estimativa do número de movimentos necessários retornando o número de blocos "a mais" no jogo multiplicado por 6 (porque 6 é o número máximo de movimentos, considerando um caminho livre, entre 2 blocos, cada um em cantos opostos).

# Algoritmos de Pesquisa

Neste trabalho foram implementados todos os algoritmos de pesquisa pedidos, sendo eles: pesquisa em largura; pesquisa em profundidade; aprofundamento progressivo; custo uniforme; pesquisa gulosa; Algoritmo A\*.

Na implementação de todos os algoritmos foi criada uma lista denominada ***visited*,** onde são guardados todos os nós já explorados. Em vez de utilizarmos uma fila de prioridade, como seria mais aconselhável para obter melhor eficiência, decidimos utilizar uma simples lista pois não conseguíamos assegurar a ordem correta dos nós quando se adicionavam à fila de prioridade. A lista ***visited*** é utilizada sempre que é explorado um novo nó para verificar se os seus filhos representam um estado de jogo que já tenha sido explorado. Esta, evita que se formem ciclos indesejados permitindo assim que os algoritmos sejam mais eficientes.

Os algoritmos de pesquisa em largura (***bfs***) e em profundidade (***dfs***) têm uma implementação muito semelhante, ambos possuem uma lista com nome ***queue*** que contém os nós que vão ser explorados. Cada nó contém o estado do jogo e os movimentos necessários para chegar ao estado do jogo ([***game***, ***path***]). O algoritmo é executado enquanto houver nós para explorar na lista ***queue***. Para cada nó a ser explorado é primeiramente verificado se o estado do jogo do nó corresponde a um estado final, se tal não acontecer então são calculados os seus filhos (conjunto de movimentos possíveis a partir do estado do jogo do nó pai). A única diferença entre os algoritmos de pesquisa em profundidade e de pesquisa em largura está na posição onde são inseridos os nós filhos na lista ***queue***, no caso da pesquisa em profundidade são inseridos no início da lista e na pesquisa em largura são inseridos no final.

Para o algoritmo A\* (***astar***) foi utilizado a lista ***queue*** que contém, como os outros algoritmos, os nós a analisar, sendo que os nós contém o custo estimado da solução mais barata que passa por cada nó, o estado do jogo e os movimentos necessários para chegar ao estado do jogo ([***f(n)***, ***game***, ***path***]). O algoritmo procura dentro da lista ***queue*** o melhor valor do custo estimado (***f(n)***) e explora-o calculando os seus nós filhos e o custo estimado para cada nó. O algoritmo de pesquisa gulosa (***greedy***) é todo igual ao algoritmo A\*, sendo que o único elemento que difere é o primeiro de cada nó pois a pesquisa gulosa só tem em conta o custo estimado para chegar ao objetivo e não o custo para chegar ao nó em questão.

O algoritmo do custo uniforme (***ucs***) é muito semelhante, em termos de implementação, ao algoritmo A\*. As únicas diferenças são que o primeiro elemento de cada nó é o custo para chegar ao nó em questão e que para cada nó filho o custo é calculado incrementando o custo do nó pai, visto que qualquer que seja o movimento no tabuleiro o custo deste é sempre 1.

Na implementação do algoritmo de aprofundamento progressivo (***iterative\_depth***) foi utilizada uma função com o nome ***dfs\_limit*** que faz o mesmo que o algoritmo de pesquisa em profundidade, mas os nós só são explorados até ao limite definido. O algoritmo de aprofundamento chama a função ***dfs\_limit*** incrementando o valor do limite até chegar a uma solução.

Todos os algoritmos retornam os movimentos calculados para chegar a um estado final (***path***) e o número de nós obtidos (***mem***), sendo este valor unicamente para análise dos algoritmos.

# Experiências e Resultados

TODO: Descrevendo as experiências realizadas com os vários algoritmos para resolver diversos puzzles e os resultados obtidos a nível de tempo e custo da solução obtida em cada nível, por cada um dos métodos experimentados. Devem ser incluídas tabelas comparativas dos resultados obtidos na aplicação dos vários métodos aos vários puzzles (níveis do jogo) e discutidos os resultados.

# Conclusões e Perspetivas de Desenvolvimento

Em primeiro lugar, é possível concordar que o conhecimento sobre métodos de pesquisa foi consideravelmente aprofundado, mostrando-se assim o grupo satisfeito com o trabalho desenvolvido. Visto ser uma área de estudo cada vez mais desenvolvida e devido ao seu aspeto futurista, a Inteligência Artificial suscitou em nós um grande interesse pela mesma, o que também contribuiu para a realização do trabalho/estudo feito.

Vários testes foram realizados de forma a recolher informação sobre o comportamento dos métodos de pesquisa utilizados. Estes foram feitos fazendo variar o nível de dificuldade do puzzle e a heurística (se fosse o caso) para cada um dos algoritmos. A análise efetuada permitiu estudar os aspetos que consideramos mais importantes como o tempo de execução, o espaço ocupado e o custo (movimentos executados).

Passando à individualização dos algoritmos foi possível concluir o seguinte:

- Como seria expectável, a pesquisa em largura chega à solução ótima, apesar de demorar mais tempo, isto pode ser verificado nos resultados apresentados.

- A\* é o algoritmo privilegiado dado que se apresenta como o melhor algoritmo, efetivamente é o que acontece quando utilizamos a terceira heurística criada.

- O algoritmo de pesquisa em profundidade apesar de se apresentar como um método rápido não é eficaz e nos testes feitos quase nunca chega à melhor solução.

- Dado que o custo de cada movimento é sempre igual, e apesar de não fazer muito sentido no contexto do nosso problema, foram igualmente feitos testes para o método de custo uniforme.

- Com a pesquisa gulosa nem sempre se chega à melhor solução, mas podemos considerar que o tempo de execução é bastante rápido, como é esperado este algoritmo procura explorar sempre o nó com melhor heurística.

- Quanto ao Aprofundamento Iterativo é possível verificar que corresponde ao esperado pois os valores de tempo confirmam que é bastante mais rápido que a pesquisa em largura e que apresenta sempre a melhor solução possível.

De modo geral acreditamos que cumprimos com os requerimentos do trabalho, as experiências efetuadas e os resultados obtidos refletem um estado positivo do trabalho desenvolvido.

##### Referências Bibliográficas

1. Rosetta Code, “15 Puzzle Game", last updated March 2019, [online], available at: <https://rosettacode.org/wiki/15_Puzzle_Game>, consulted on March 2019.
2. Milan Pecov, “15-puzzle", 2013, [online], available at: <https://github.com/MilanPecov/15-Puzzle-Solvers>, consulted on March 2019.