Resolução do *Roll the Block* utilizando Métodos de Pesquisa em Linguagem JAVA (Tema 5 / Grupo 53)

Manuel Monteiro (201504445)
MIEIC
FEUP
Porto, Portugal
up201504445@fe.up.pt

Diogo Moreira (201504359)
MIEIC
FEUP
Porto, Portugal
up201504359@fe.up.pt

Ao longo das últimas semanas, o grupo tem estado a desenvolver um programa que consiga a melhor solução possível para alguns níveis do jogo Roll the Block em IART.
Por causa de algumas limitações, como por exemplo, a grande quantidade de níveis e o tempo curto definido para desenvolver este projeto, restringimos o nosso trabalho aos primeiros níveis, tentando obter um projeto robusto e eficiente.

I. Introdução

No âmbito da unidade curricular Inteligência Artificial do Mestrado Integrado em Engenharia Informática e Computação, foi-nos sugerida a elaboração de um projeto que conseguisse encontrar soluções para certos jogos. Esse jogo foi selecionado pelo grupo dentro de um leque de várias opções disponibilizadas pelos docentes.
A escolha do *Roll the Block* baseou-se no estilo que o prescrevia. A jogabilidade simples e a combinação de jogadas possíveis tornaram-no numa boa opção e a motivação do grupo para a sua resolução foi forte.
Este jogo tem algumas características particulares que o tornam um desafio extremamente interessante, tal como o facto de ter duas unidades de altura e uma de base, o que faz com que a movimentação do objeto seja mais complexa.
O objetivo deste trabalho é a aplicação dos primeiros conceitos interiorizados nas aulas teóricas e desenvolvidos nas aulas práticas da unidade curricular.

II. Descrição do Problema

O jogo *Roll the Block* é um jogo do estilo *puzzle*, que é jogado já há alguns anos em todas a plataformas mais usadas como android, iOS e até PC.

Fig. 1. Imagem do jogo

Regras

Este jogo é estilo 3D e é constituído pelas seguintes regras:

a) Movimentos possíveis: só existem quatro movimentos possíveis - norte, sul, este, oeste.
b) Movimentos proibidos: sempre que o bloco é movido de modo a ficar com uma ou duas unidades de fora do tabuleiro, ele cai, o que faz com que o utilizador tenha de recomeçar o nível.

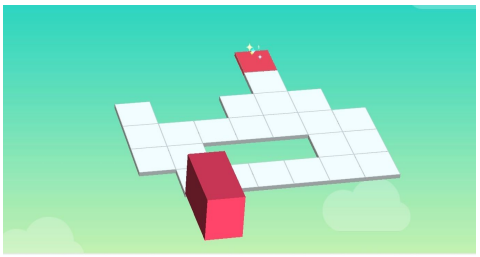

Fig. 2. Exemplo de utilizador a perder

Objetivo

O objetivo do jogo é chegar à posição assinalada a vermelho. No entanto, o bloco tem de estar na posição vertical, o que dificulta exponencialmente o jogo, não bastando encontrar apenas o percurso mais rápido. Se assim fosse bastava aplicar o Dijkstra ou A*, mas é necessário tomar também em consideração a orientação final.

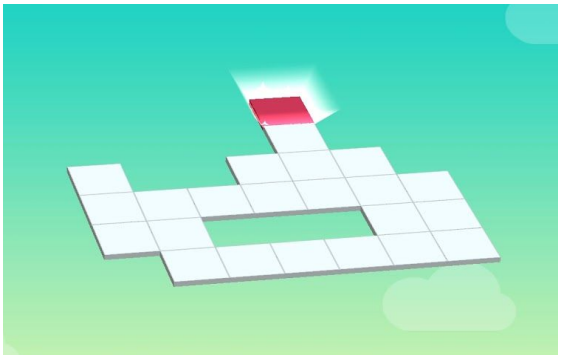

Fig. 3. Exemplo de utilizador a ganhar

III. Formulação do Problema

O estado de jogo tem de ser caracterizado por uma matriz com o tabuleiro em que a largura e comprimento são determinados pela maior distância entre os quadrados nos extremos, tanto na horizontal como na vertical, mais dois em cada lado para espaços vazios. Cada posição da matriz pode assumir os seguintes valores:

- 0 - Célula vazia;
- 1 - Célula alcançável;
- 2 - Célula final;
- 3 - Bloco.

Temos também o estado do bloco que, apesar de o conseguirmos deduzir pelo tabuleiro, facilita termos uma variável de orientação a ele dedicada, como:

- 0 - Vertical;
- 1 - Este;
- 2 - Sul;
- 3 - Oeste;
- 4 - Norte.

Estado inicial

O estado inicial começa com uma célula a {3}, posição inicial do bloco na vertical, uma célula a {4}, posição final a ser atingida e o resto das células são ou vazias ou alcançáveis.

Testes

- Minimizar número de movimentos;
- Não cair em nenhuma célula vazia;
- Chegar à célula final na orientação vertical.

Operadores

Mover o bloco para norte, este, oeste ou sul:

Tabela 1
Operadores

Operador	**Pré-condição**	**Efeito**	**Custo**

Norte	Se orientação == 0{
	(Tabuleiro[x - 1][y] ∧ Tabuleiro[x - 2][y]) != 0
	}
	Se orientação != 0{
	Tabuleiro[x - 1][y] != 0
	}
		Se orientação == 0{
		Tabuleiro[x - 1][y] ∧ Tabuleiro[x - 2][y] = 3
		}
		
		Se orientação = 1{
		Tabuleiro[x-1][y] = 3
		}
		
		As células anteriores são repostas ao estado inicial
			10

Sul	Se orientação == 0{
	(Tabuleiro[x + 1][y] ∧ Tabuleiro[x + 2][y]) != 0
	}
	Se orientação != 0{
	Tabuleiro[x + 1][y] != 0
	}
		Se orientação == 0{
		Tabuleiro[x + 1][y] ∧ Tabuleiro[x + 2][y] = 3
		}
		
		Se orientação != 0{
		Tabuleiro[x + 1][y] = 3
		}
			10

		As células anteriores são repostas ao estado inicial	
Este	Se orientação == 0{ (Tabuleiro[x][y + 1] ∧ Tabuleiro[x][y + 2]) != 0 } Se orientação != 0{ Tabuleiro[x][y + 1] != 0 }	Se orientação == 0{ Tabuleiro[x][y + 1] ∧ Tabuleiro[x][y + 2] = 3 } Se orientação != 0{ Tabuleiro[x][y + 1] = 3 } As células anteriores são repostas ao estado inicial	10
Oeste	Se orientação == 0{ (Tabuleiro[x][y - 1] ∧ Tabuleiro[x][y - 2]) != 0 } Se orientação != 0{ Tabuleiro[x][y - 1] != 0 }	Se orientação == 0{ Tabuleiro[x][y - 1] ∧ Tabuleiro[x][y - 2] = 3 } Se orientação != 0{ Tabuleiro[x][y - 1] = 3 } As células anteriores são repostas ao estado inicial	10

Nota: Se o bloco estiver na vertical o custo é metade do custo original.

IV. Trabalho Relacionado

Como mencionado, iremos usar o **A***. Conseguimos encontrar alguns projetos semelhantes ao que agora apresentamos, embora todos eles aplicados ao jogo *Bloxorz* (versão original do *Roll the Block*). Os projetos são desenvolvidos em Python ou Java, maioritariamente. Os projetos encontrados mais relevantes foram os seguintes:

- https://github.com/phamquiluan/BloxorzSolver
- https://github.com/thanhhocse96/bloxorz-python-solver/tree/master/src
- https://github.com/alppboz/bloxorz_solver?fbclid=IwAR34G9SnXAzBzEnAZxFkrQ7V-RpIVtkLiD-29zIuRV8GtpJLy-Mb2F2CbKc

V. Implementação do Jogo

A nossa abordagem para encontrar resoluções para o *Roll the Block* passou por enquadrar o mapa do jogo, a posição do bloco (representada por 3), estado inicial, estado final (representada por 4), espaços vazios (representada por 0) e espaços disponíveis (representada por 1), numa matriz (Board.java), e orientação atual do bloco numa estrutura de dados. De seguida, corremos os algoritmos de pesquisa que se ajustam ao nosso problema (AStar.java, BFS.java e DLS.java).

O bloco pode então só ser movido em quatro direções: norte, este, sul, oeste, nunca podendo cair num espaço vazio. Podemos ter esta informação em consideração, de modo a reduzir a memória e tempo utilizado para correr os algoritmos

Todos os movimentos possíveis para uma dada posição do bloco são calculados numa classe à parte (utils.java). Esta classe contém uma função que devolve os nós adjacentes, se for possível o bloco deslocar-se nessa direção.

Utilizando a informação fornecida pela classe utils.java decidimos então se devemos adicionar o node à nossa stack ou priorityQueue, dependendo do algoritmo utilizado.

Para sabermos se o algoritmo teve sucesso em encontrar o caminho até à posição final só precisamos de uma condição, cuja função é apenas a de verificar se o bloco está na posição vertical e o node se encontra na posição igual à do node final.

Como cada Node contém uma variável denominada parent do tipo Node, podemos reconstruir o caminho percorrido pelo bloco de forma recursiva.

VI. Algoritmos de Pesquisa

Os algoritmos de pesquisa utilizados foram o A*, o DLS (Depth Limited Search) e o BFS (Breadth First Search).
O algoritmo do Custo-Uniforme demonstrou ter uma implementação demasiado semelhante ao A* neste problema, por isso não foi implementado.

Para o algoritmo DLS, o mais simples no que toca à implementação, temos apenas de correr um ramo, começando pelo ramo mais à esquerda até que o nível seja igual ao nível máximo escolhido pelo utilizador. Se uma solução for encontrada então guardamos esse nó numa variável à parte, denominada de bestNode. Assim que o nível máximo for atingido passamos ao Node que se encontra no nível igual àquele onde se começou a desenvolver esta iteração, repetindo o processo até não existirem mais Nodes nesse nível. Passamos então para o nível abaixo, repetindo mais uma vez o todo o processo até não existirem mais Nodes disponíveis acima do nível máximo.
Expandimos todos os Nodes excepto se o Node seguinte atual for igual ao parent do Node atual ou se o Node seguinte for negado pela classe Utils.java, o que significa que não é um movimento permitido.

Para o algoritmo BFS, ao contrário do DLS, em vez de percorrermos um ramo até o nível limite, percorremos todos os Ñodes num nível e só depois avançamos para o seguinte. Esta implementação é vantajosa pois assim que encontrar uma solução sabemos que é, sem dúvida, uma solução ótima já que estará no menor nível possível (= menor número de moves possível).
Expandimos todos os Nodes, excepto se o Node seguinte atual existir na liste de Nodes vistos ou se o Node seguinte for negado pela classe Utils.java, o que significa que não é um movimento permitido.

Para o algoritmo A*, ao contrário dos outros algoritmos, usamos uma priorityQueue onde os Nodes são organizados por menor custo. O custo é calculado pela soma da distância do Node à posição final mais o custo de movimento total até essa posição. Dependendo da orientação atual variamos o custo do movimento, sendo que estes custos são duplicados quando o bloco está deitado, o que faz com que só seja possível mover-se um quadrado para qualquer lado, enquanto que, se estiver na orientação vertical, pode mover-se dois quadrados com um só movimento para qualquer posição adjacente.
Assim que se chegar à posição final na orientação vertical temos a solução ótima.

VII. Experiências e Resultados

Variáveis usadas durante os testes:
DLS - MaxSearchDepth = 14
A* - VerticalMoveCost = 5,
HorizontalMoveCost = 10

Tabela 2
Resultados do Nível 1

Nível 1	**Nodes** (uni)	**Tempo** (ms)
DLS	2399543	452
BFS	133	6
A*	53	29

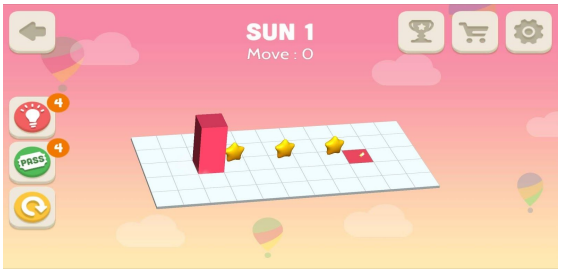

Fig. 4. Nível 1

Tabela 3
Resultados do Nível 2

Nível 2	**Nodes** (uni)	**Tempo** (ms)
DLS	53806	79
BFS	35	3
A*	29	25

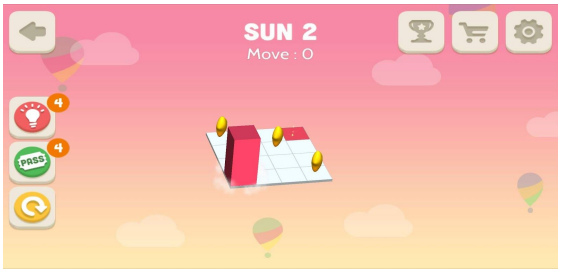

Fig. 5. Nível 2

Tabela 4
Resultados do Nível 3

Nível 3	**Nodes** (uni)	**Tempo** (ms)
DLS	2375	11
BFS	23	4
A*	26	27

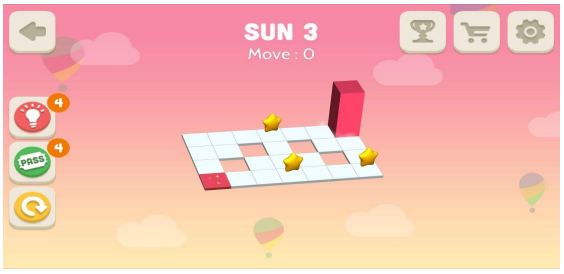

Fig. 6. Nível 3

Tabela 5
Resultados do Nível 4

Nível 4	**Nodes** (uni)	**Tempo** (ms)
DLS	229	5
BFS	18	5
A*	23	26

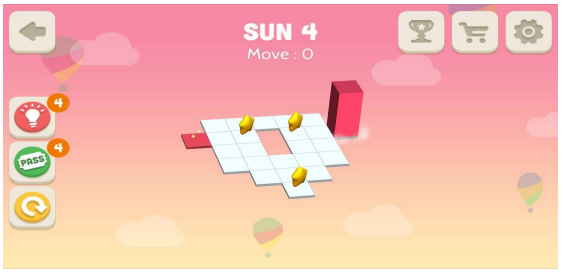

Fig. 7. Nível 4

Tabela 6
Resultados do Nível 5

Nível 5	**Nodes** (uni)	**Tempo** (ms)
DLS	6435	23
BFS	60	5
A*	74	27

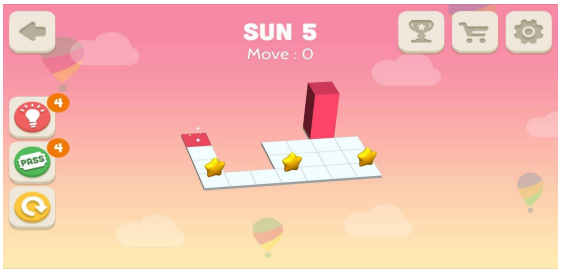

Fig. 8. Nível 5

Tabela 7
Resultados do Nível 6

Nível 6	**Nodes** (uni)	**Tempo** (ms)
DLS	8026	24
BFS	91	4
A*	61	27

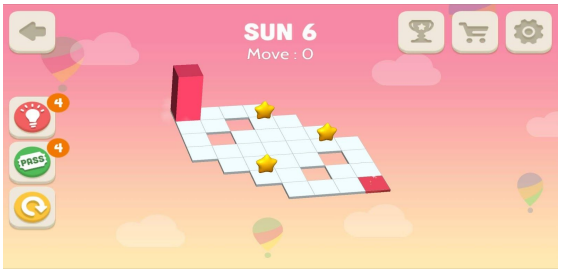

Fig. 9. Nível 6

Tabela 8
Resultados do Nível 7

Nível 7	**Nodes** (uni)	**Tempo** (ms)
DLS	505	5
BFS	23	5
A*	28	28

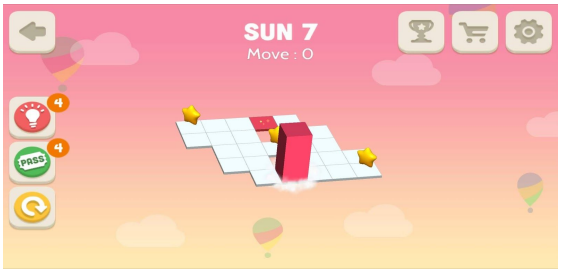

Fig. 10. Nível 7

Tabela 9
Resultados do Nível 8

Nível 8	**Nodes** (uni)	**Tempo** (ms)
DLS	117	4
BFS	18	4
A*	28	27

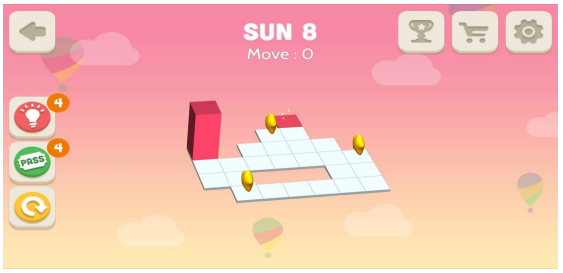

Fig. 11. Nível 8

Tabela 10
Resultados do Nível 9

Nível 9	**Nodes** (uni)	**Tempo** (ms)
DLS	3190	13
BFS	35	4
A*	45	27

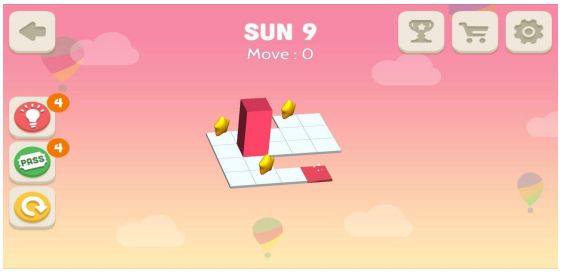

Fig. 12. Nível 9

Tabela 11
Resultados do Nível 10

Nível 10	**Nodes** (uni)	**Tempo** (ms)
DLS	2765	11
BFS	136	6
A*	144	28

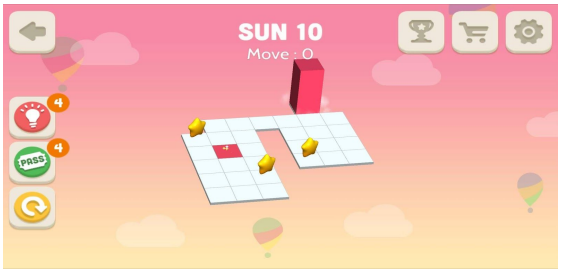

Fig. 12. Nível 10

Tabela 12
Resultados do Nível X

Nível 10	**Nodes** (uni)	**Tempo** (ms)
DLS	--	--
BFS	2090	27
A*	1004	38

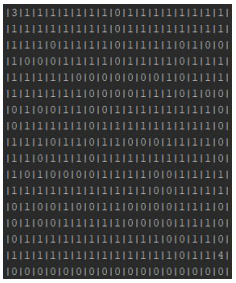

Fig. 14. Nível X

VIII. Conclusões e Perspetivas de Desenvolvimento

As conclusões a que chegamos após o desenvolvimento deste trabalho podem ser interpretadas de várias formas. Quanto aos algoritmos, podemos basear-nos no algoritmo BFS sempre que temos poucas variáveis em jogo. Como até ao último nível da primeira fase deste jogo os mapas são relativamente pequenos, o algoritmo BFS não tem o seu resultado esperado, que seria demorar muito mais tempo que o A* e ocupar muita mais memória. Esses resultados são obtidos na mesma, mas com diferenças praticamente desprezíveis em relação ao A*. No entanto, sabemos que a longo prazo ou em mapas maiores o A* será sempre a nossa melhor solução.
O algoritmos DLS mostrou-se muito pouco prático para este problema, visto que cada nível demonstra ter poucas soluções e o objetivo do jogo é encontrar a solução ótima para cada nível.

Nós estamos certos de que podemos melhorar ainda mais a heurística usada no A*. No futuro, gostaríamos de explorar mais formas para conseguir diferenciar os pesos entre os movimentos de maneira a expandir um menor número de nós, chegando mais rapidamente e eficientemente à solução ótima.

Referências Bibliográficas

[2] "Amit's A* Pages", 2019, [online], available at: http://theory.stanford.edu/~amitp/GameProgramming/ , consulted on March 2019.
[3] Phanmquiluam, "BloxorzSolver", 2018, [online], available at: https://github.com/phamquiluan/BloxorzSolver, consulted on March 2019.
[4] Thanhhocse96, "Bloxorz-Python-Solver", 2018, [online], available at: https://github.com/thanhhocse96/bloxorz-python-solver, consulted on March 2019.
[5] Thanhhocse96, "Bloxorz-Python-Solver", 2018, [online], available at: https://github.com/alppboz/bloxorz_solver, consulted on March 2019.

[6] Alppboz, “Bloxorz_Solver”, 2018, [online], available at: https://github.com/alppboz/bloxorz_solver, consulted on March 2019.