IART – Checkpoint 1

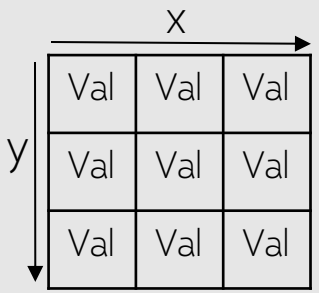
Alexandre Carqueja – 201705049
Henrique Santos – 201706898
Tiago Alves - 201603820

Folding Blocks – SinglePlayer Game

- "Folding Blocks" é um jogo tipo puzzle, em que o objetivo é desdobrar os blocos de modo a preencher todo o tabuleiro
- Quando se desdobra um bloco, este é estendido com as mesmas dimensões na direção do movimento.
- Podem existir vários blocos de partida, sendo estes distinguidos pela cor.
- Apenas se pode desdobrar o objeto caso este não vá sobrepor nenhum outro bloco ou saia do tabuleiro após o movimento.

Formulação do Problema

- Representação do estado:
  - Matriz x por y
  - Val = {-1,0,1,...}
    - -1 -> null space
    - 0 -> empty space
    - Val > 0 -> some color
- Estado inicial:
  - Matriz começa com pelo menos um Val = 0, e pelo menos um Val > 0.
- Estado final:
  - Se não houver espaços a 0, jogador ganha
  - Se houver espaços a 0 e não for possível fazer mais operações, o jogador perde

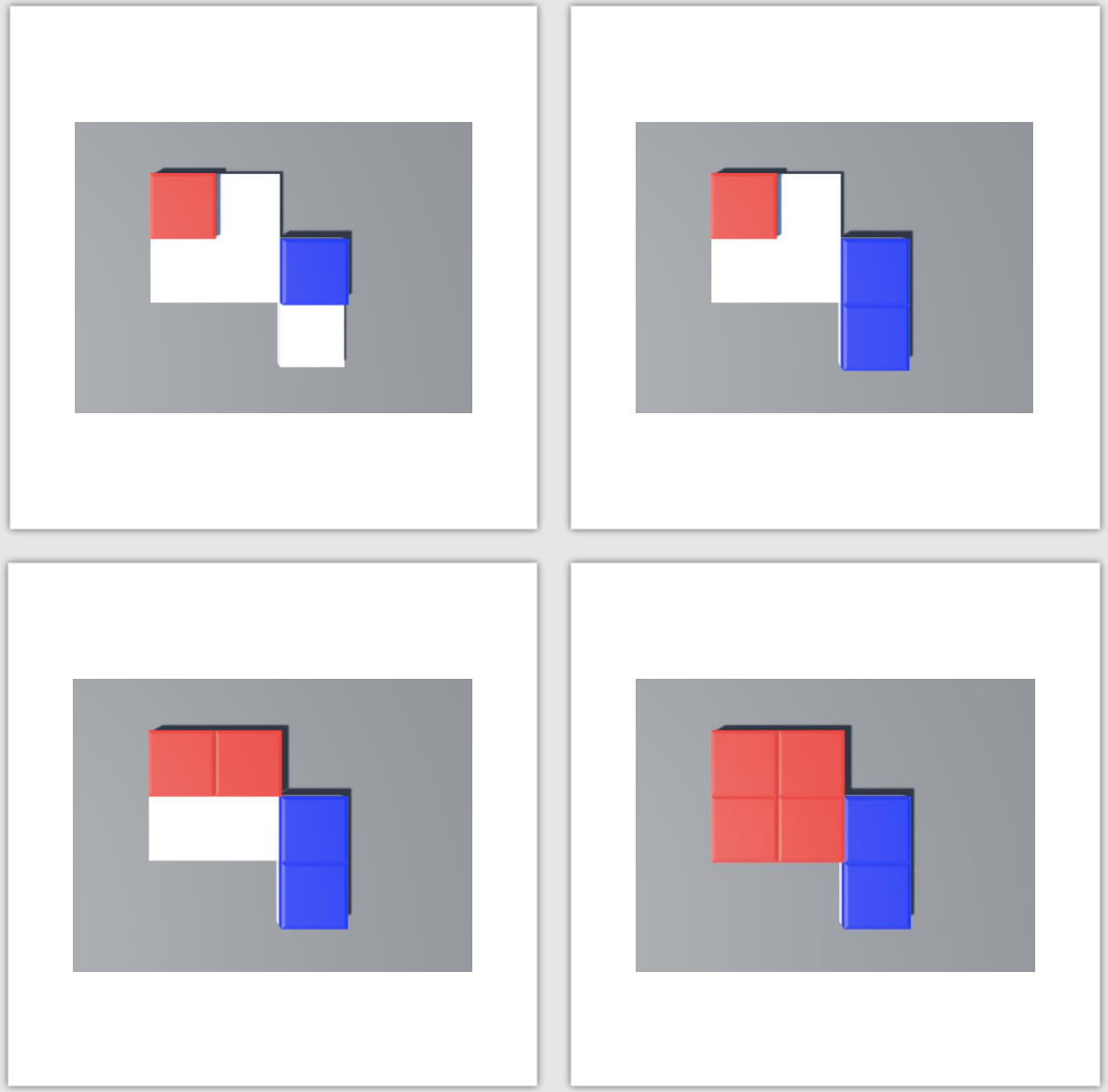


1	0	-1
0	0	2
-1	-1	0

1	1	-1
1	1	2
-1	-1	2

- Operadores:

Nome	Pré-Condições	Efeitos
Flip Val up	Val > 0, axis = min(y) where Matrix[x][y] == Val, ∀piece ∈ Matrix[piece.x,2*axis-piece.y - 1] == 0	∀piece, Matrix[piece.x,2*axis-piece.y - 1] = Val
Flip Val down	Val > 0, axis = max(y) where Matrix[x][y] == Val, ∀piece ∈ Matrix[piece.x,2*axis-piece.y + 1] == 0	∀piece, Matrix[piece.x,2*axis-piece.y + 1] = Val
Flip Val right	Val > 0, axis = max(x) where Matrix[x][y] == Val, ∀piece ∈ Matrix[2*axis-piece.x - 1,piece.x] == 0	∀piece, Matrix[2*axis-piece.x - 1, piece.y] = Val
Flip Val left	Val > 0, axis = min(x), where Matrix[x][y] = Val, ∀piece ∈ Matrix[2*axis-piece.x + 1,piece.y] == 0	∀piece, Matrix[2*axis-piece.x + 1, piece.y] = Val



Trabalho desenvolvido

- Implementada uma interface gráfica aproximada da do jogo original.
- Implementados todos os métodos de pesquisa propostos, com adição de dois algoritmos gananciosos.
- Implementado o modo de jogador Humano, com "dicas" caso o jogador esteja com dificuldade.
- Implementado o modo de computador, onde o utilizador escolhe o puzzle que quer resolvido e via que algoritmo.
- Implementada uma plataforma de testes de modo a testar exaustivamente todos os puzzles e métodos de pesquisa.
- Comparação minuciosa dos métodos de pesquisa e conclusões resultantes.

Operadores

Cada operador tem 3 fases principais:

- Descobrir o eixo de rotação
- Verificar se o simétrico de cada tile em relação ao eixo de rotação é válido, retornando falso caso algum destes não seja possível. Nesta fase guarda-se também a posição de todos os simétricos dos tiles da cor que se pretende mover.
- Função usada para calcular simétrico:
  - i - 2 * (i - rotationAxis) - 1;
- Por fim, após ter verificado que é um movimento válido, percorrer a lista de posições e alterar a cor de todos os simétricos para a cor pretendida.

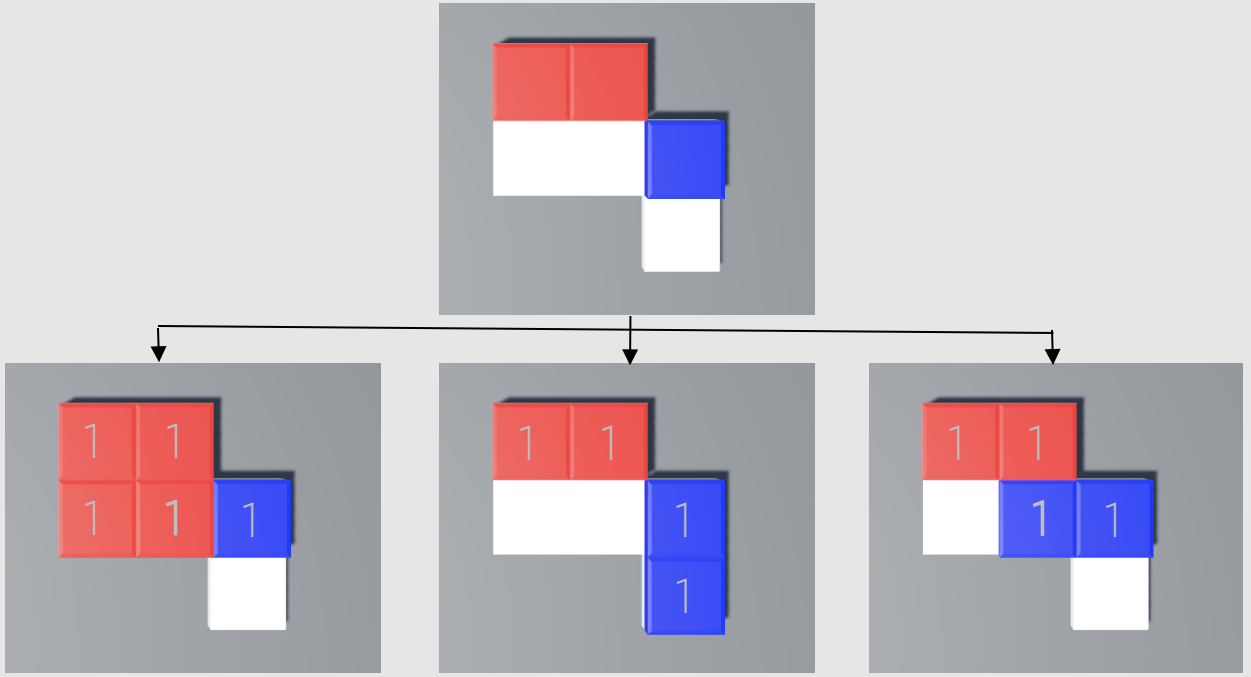
Visto que os operadores modificavam o puzzle que se pretendia modificar, durante os processos de pesquisa era necessário uma maneira de passar o mesmo objeto para várias funções.

Inicialmente sempre que se pretendia executar um operador, fazia-se uma cópia de modo a não modificar o original. No entanto, chegou-se à conclusão que utilizar um operador de undo, que revertesse a operação de volta atingia resultados consideravelmente melhores.

Heurísticas

- Simple Fill:

Cada move é avaliado pela percentagem do nível preenchido após este; é uma heurística gulosa que prefere moves maiores, mesmo que estes impossibilitem a resolução do puzzle.

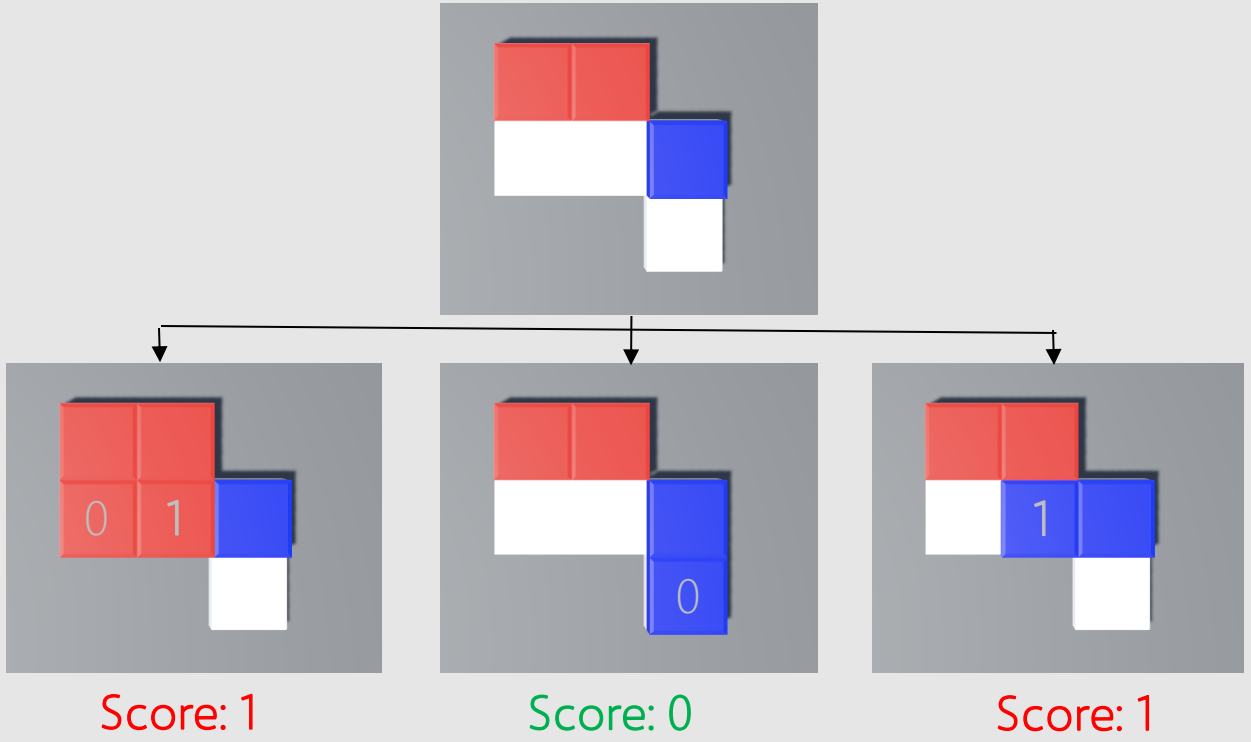


Formulação do Problema Cont./ Heurística

- Unique-First:

Para todos os moves possíveis, expandem-se as cores de forma a dar a cada tile um score que indica quantas cores conseguem expandir para este numa única expansão. Cada move é avaliado como a soma dos scores dos tiles envolvidos no move. Em caso de empate, expandem-se os moves que sucederiam os moves empatados, e repete-se este desempate até se encontrar uma sequência de moves com o menor número de "colisões" com outras cores possível.

No final será escolhido o estado com menor pontuação.

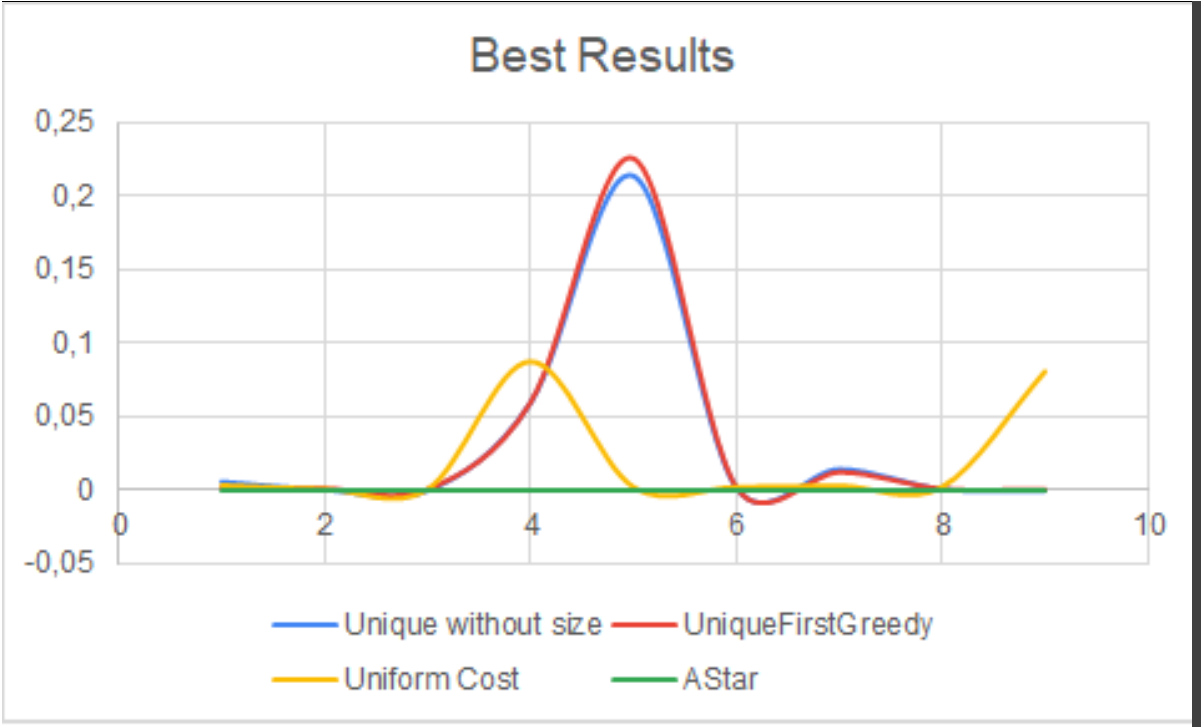


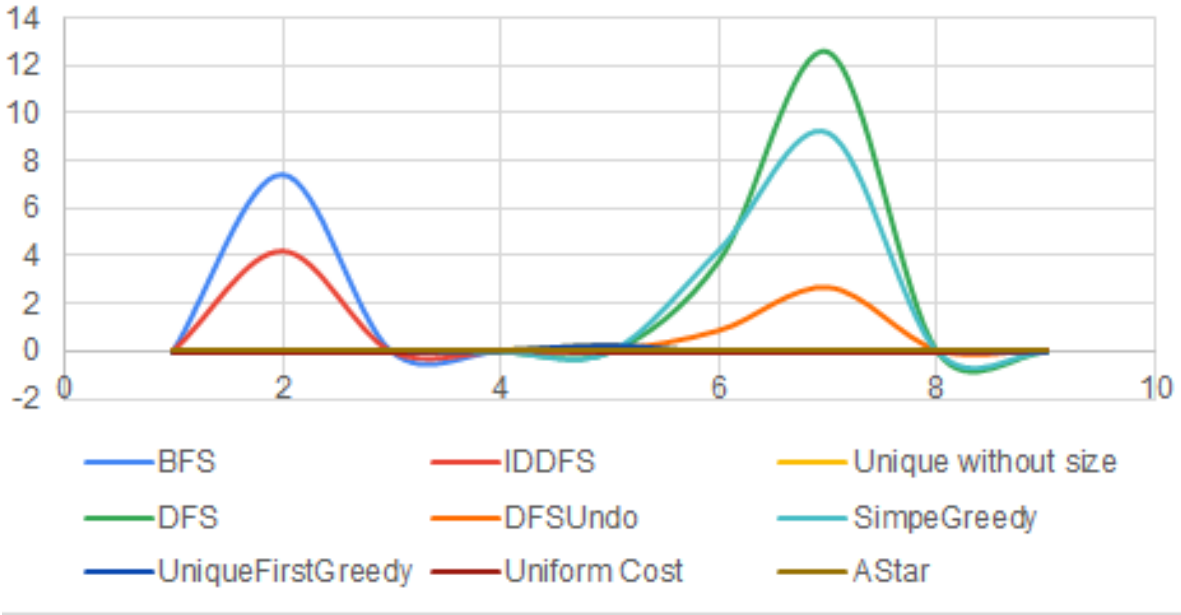
Tempo demorado(s)

Puzzle	BFS	IDDFS	Simple Greedy	DFS	Unique Size	Unique Greedy	Uniform Cost	A*
1	0,0055	0,0142	0,002	0,001	0,006	0,0013	0,0028	0.0065
2	7,4598	4,259	0,0862	0,0042	0,0005	0,0006	0,0002	0,0005
3	NC	52.079	0	0,0001	0	0	0	0
4	NC	NC	0	0	0,0595	0,0592	0,0872	0,032
5	NC	NC	NC	NC	0,2135	0,2257	0,0016	0,005
6	NC	NC	4,2606	0,8317	0,002	0,0016	0,0016	0,001
7	NC	NC	9,1458	2,6392	0,0145	0,0118	0,0024	0,002
8	NC	NC	0,0016	0,002	0,001	0,0004	0,0016	0,0005
9	NC	NC	NC	NC	NC	NC	0,0804	0,001

Nº de nós visitados

Puzzle	BFS	IDDFS	Simple Greedy	DFS	DFS Undo	Unique Greedy	Uniform Cost	A*
1	344	170	7	7	7	3	5	3
2	150680	189247	1732	400	400	23	10	6
3	NC	7447582	10	10	10	6	8	6
4	NC	NC	12	12	12	1905	2441	999
5	NC	NC	NC	NC	NC	6003	12	8
6	NC	NC	58608	56839	56839	7	22	7
7	NC	NC	199535	199346	199346	349	48	8
8	NC	NC	12	12	12	7	15	7
9	NC	NC	NC	NC	NC	NC	2014	7





Conclusões

- O algoritmo que atingiu os melhores resultados foi o A*, que combina a Heurística de Unique-First com a Heurística de Simple Fill.
- O custo uniforme, apesar de ser uma pesquisa não informada, vê para cada puzzle os estados seguintes, tendo atingido excelentes resultados.
- Apesar de usarem a mesma heurística, o Uniform Cost consegue em geral ter melhores resultados que o Unique Greedy devido à sua priority queue.
- Devido ao facto de a solução estar sempre no fim da árvore e devido à utilização de operadores mais eficientes comparativamente aos outros algoritmos, o DFS teve em geral bons resultados
- Visto que o BFS e o IDDFS são algoritmos que resultam melhor quando a solução está perto da raiz, estes apenas conseguiram resolver os primeiros puzzles.
- A Heurística de preencher a maior quantidade de tabuleiro possível teve resultados ligeiramente melhores que o DFS, tendo melhorado substancialmente o A*
- Sendo o último nível o com mais estados possíveis, apenas o A* e o custo uniforme conseguiram resolver.

Referências

Embora não tenhamos encontrado nenhum código fonte/artigo referente exatamente ao nosso jogo, encontramos alguns artigos em que são mencionadas heurísticas que achamos pertinentes aplicar na nossa resolução:

- http://www.pvv.ntnu.no/~spaans/spec-cs.pdf
- https://www.aaai.org/ocs/index.php/SOCS/SOCS16/paper/viewFile/13968/13258
- http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.175.268&rep=rep1&type=pdf
- https://towardsdatascience.com/sliding-puzzle-solving-search-problem-with-iterative-deepening-a-d7e8c14eba04

Tecnologias Usadas

As tecnologias usadas neste trabalho foram:

- Unity - Interface Gráfica
- C# - Lógica do jogo e métodos de pesquisa