Resolução do Unblock me - Grupo 23

Utilizando Métodos de Pesquisa em Linguagem java

António Cruz (up201603526)  
FEUP, MIEIC up201603526@fe.up.pt

Helena Montenegro (up201604184)

FEUP, MIEIC  
up201604184@fe.up.pt

Juliana Marques (up201605568)  
FEUP, MIEIC  
up201605568@fe.up.pt

*Resumo* - Este artigo contém a abordagem de desenvolvimento de algoritmos de pesquisa na linguagem java para resolver o jogo para android *Unblock Me*. Foram implementados algoritmos de pesquisa em largura, em profundidade, aprofundamento progressivo, gananciosa e A\*, dos quais se concluiu que o A\* é o mais eficiente, devolvendo uma solução ótima e em pouco tempo de execução.

Keywords - Inteligência Artificial, Algoritmos de pesquisa

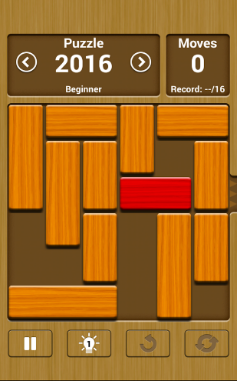
# Introdução

O projeto desenvolvido tem como objetivo o desenvolvimento das aptidões e conhecimentos obtidos no decorrer da cadeira de Inteligência Artificial do MIEIC.

Foram implementados vários algoritmos de pesquisa com o objetivo de encontrar a solução ao jogo “Unblock Me”, cujo objetivo é mover blocos de modo a que o bloco vermelho chegue à saída. Neste artigo será descrito o jogo em questão, como foi formulado como um problema de pesquisa, como o problema foi resolvido e os seus resultados.

# Descrição do Problema

O *Unblock Me* é um jogo que foi lançado em 2009 por *Kiragames Co., Ltd*. O seu objetivo é remover o bloco vermelho do tabuleiro. Os blocos são posicionados no tabuleiro de forma vertical ou horizontal; um bloco horizontal só se pode mover para a esquerda ou para a direita enquanto os blocos verticais apenas se podem mover para cima e para baixo. O jogador deve mover os blocos de forma a conseguir arranjar um caminho livre para o bloco vermelho sair.



# Formulação do Problema

## Representação do Estado

O tabuleiro é uma matriz em que as casas vazias são representadas por zeros, os limites do tabuleiro são representados por -2, a saída é representa por -1, o bloco que quer sair é representado por 1 e os restantes blocos têm identificadores inteiros superiores a 1 em que identificadores pares estão orientados horizontalmente e ímpares verticalmente.

## Estado inicial

O estado inicial dependerá do nível em causa representando a matriz de acordo com as regras explicadas previamente.

## Teste Objetivo

O estado final em que se considera o puzzle resolvido será quando na matriz o elemento 1 está ao lado do elemento -1 (ao lado da saída), ou seja, na mesma linha numa posição consecutiva.

## Operadores

*Nome:*

As operações possíveis para cada bloco são: Andar para cima, para baixo, para a esquerda ou para a direita.

*Pré-Condições:*

Apenas os blocos com identificador único par se podem mover para a esquerda e para a direita. Por outro lado, apenas os blocos com identificador único ímpar se podem mover para cima e para baixo.

Um bloco apenas se pode mover, se nessa direção existir um espaço vazio (0 na matriz).

*Efeitos:*

Com um movimento a matriz será atualizada de forma a que o bloco movido ocupe os 0's na matriz por o seu identificador e deixando os espaços ocupados previamente com 0's.

*Custo:*

Cada movimento de um bloco para um lugar adjacente terá o custo de um movimento.

## Custo da Solução

O custo da solução obtida será composto pelo custo de pesquisa associado à descoberta desta, juntamente com o número de jogadas efetuadas.

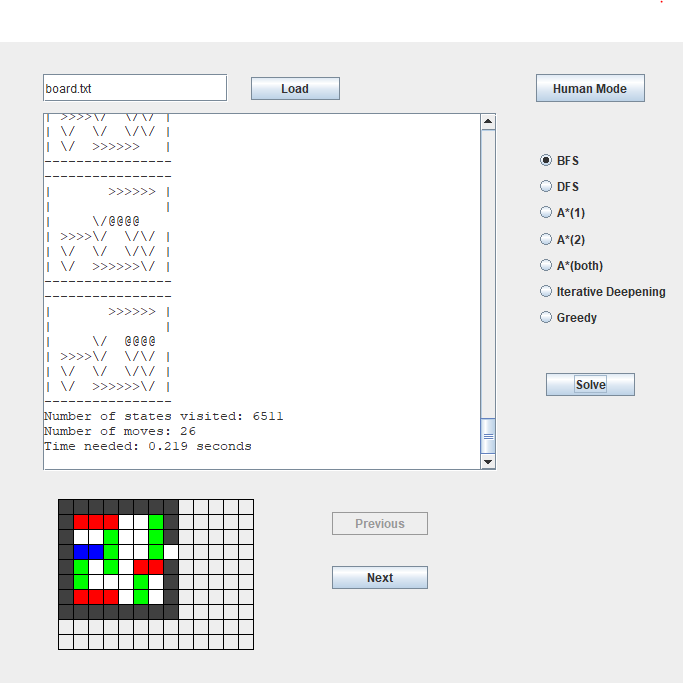
# Trabalho Relacionado

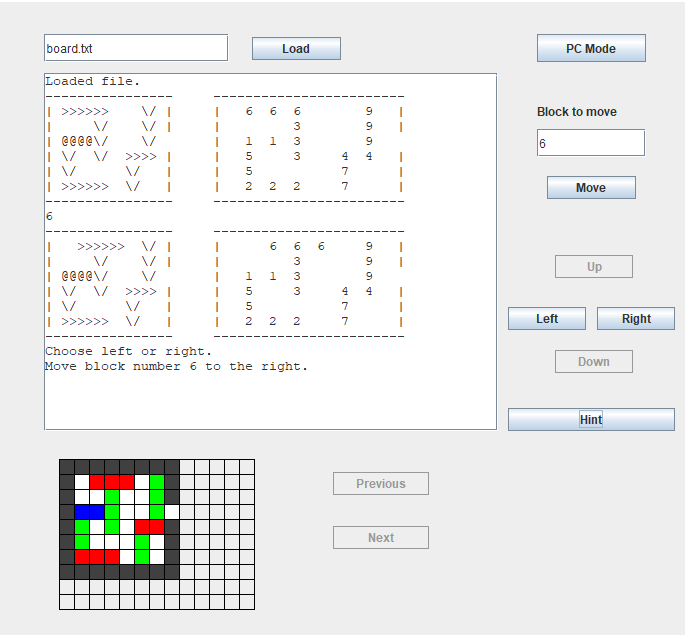
O jogo *Unblock Me* é uma nova versão de um jogo chamado *Rush Hour*, onde outros estudantes já exploraram as diferentes formas de encontrar soluções para os diferentes níveis através de métodos de pesquisa.

Na resposta a esta questão no *Stack Overflow* [1] podemos encontrar uma implementação de uma pesquisa em largura feita em Java juntamente com uma explicação do algoritmo utilizado para a resolução do mesmo problema que pretendemos resolver.

# Implementação do Jogo

O projeto foi desenvolvido na linguagem java. O projeto contém uma representação gráfica de fácil utilização que aparece quando se corre o ficheiro *gui/UI.java*. Tem também uma representação em texto na consola que é acessível ao correr *logic/Main.java*.





Para além dos algoritmos, foi implementada a possibilidade de o utilizador poder jogar, podendo pedir pistas ao programa, que as devolve usando o algoritmo A\* com a heurística definida pela soma da distância do bloco vermelho à saída com o número de blocos a bloquear a saída. Para tal, o utilizador tem que escolher o bloco que quer mover. Se este bloco conseguir mover-se em dois sentidos, é pedido o sentido do movimento do bloco ao utilizador.

O estado do tabuleiro é representado na classe *Board*, a qual tem como atributos:

* o tabuleiro em si, matriz de inteiros,
* a profundidade do tabuleiro,
* as funções de distância à solução e distância percorrida desde o tabuleiro inicial (necessárias aos algoritmos de pesquisa),
* uma *string* que evidencia qual o algoritmo de pesquisa a ser utilizado,
* o pai do board, isto é, o objeto do tipo *Board* que o gerou,
* os seus sucessores, que são gerados no decorrer do algoritmo,
* e os blocos que contêm.

Entre as funções mais importantes que esta classe contém, consta:

* uma função que permite verificar se o tabuleiro é um estado final (*is\_final()*),
* uma função que permite expandir o tabuleiro, ao verificar para cada bloco se este se pode mover e em que sentidos (*generate\_successors()*),
* funções para mostrar o tabuleiro (*print\_board()* e *print\_board\_for\_human()*),
* e a função *calculate\_h()* permite descobrir a distância desde um tabuleiro ao tabuleiro final, tendo como heurísticas disponíveis, o número de blocos entre o bloco vermelho e a saída, a distância do bloco à saída e a soma de ambas estas heurísticas.

Para obter a sequência de tabuleiros gerada pelos algoritmos de pesquisa, existe a função *generate\_sequence()* na classe *Search*, que vai verificando os pais dos tabuleiros começando no tabuleiro final e colocando-os numa lista pela ordem correta.

# Algoritmos de Pesquisa

As classes relativas à pesquisa derivam da classe *Search* que tem como atributos uma lista de tabuleiros por expandir e uma lista de tabuleiros já expandidos.

Todos os algoritmos começam por colocar o tabuleiro inicial na lista de tabuleiros por expandir. Os tabuleiros já expandidos são utilizados para comparação com os tabuleiros em análise nos algoritmos.

Em quase todos os algoritmos, à exceção da pesquisa com aprofundamento progressivo, são ignorados os tabuleiros que já foram analisados. Na pesquisa com aprofundamento progressivo, não é possível ignorar estados repetidos porque, apesar de os tabuleiros serem repetidos, a sua profundidade é diferente e um tabuleiro encontrado primeiro pode não chegar a uma solução tendo em conta a profundidade definida como limite, mas pode encontrar solução se for encontrado mais à frente com uma profundidade inferior ao primeiro. Deste modo, no caso deste algoritmo, são ignorados os tabuleiros que já tenham sido explorados e cuja profundidade é superior à do tabuleiro já explorado.

## Pesquisa em largura

A pesquisa em largura está definida no ficheiro *logic/BFS.java*. Enquanto a lista de tabuleiros por expandir não estiver vazia, expande-se o primeiro elemento e verifica-se se as expansões originam um tabuleiro final. Caso seja encontrado um tabuleiro final, este é retornado, senão adicionam-se as expansões no final da lista de tabuleiros para expandir. Se a lista de tabuleiros ficar vazia, significa que não foram encontradas soluções.

## Pesquisa em profundidade

A pesquisa em profundidade está definida no ficheiro *logic/DFS.java*. Enquanto a lista de tabuleiros por expandir não estiver vazia, expande-se o primeiro elemento e verifica-se se as expansões originam um tabuleiro final. Caso seja encontrado um tabuleiro final, este é retornado, senão adicionam-se as expansões no início da lista de tabuleiros para expandir. Se a lista de tabuleiros ficar vazia, significa que não foram encontradas soluções.

## Pesquisa com aprofundamento progressivo

A pesquisa com aprofundamento progressivo está definida no ficheiro *logic/IterativeDeepening.java*. Começa com uma profundidade limitada mínima de 10 e máxima de 100. Para cada profundidade, é executado o algoritmo de profundidade limitada descrito a seguir. Enquanto a lista de tabuleiros por expandir não estiver vazia, verifica-se se o primeiro elemento já foi explorado com uma profundidade inferior e, nesse caso, ignora-se o tabuleiro ou, em caso negativo, expande-se o tabuleiro e verifica-se se as expansões originam um tabuleiro final. Caso seja encontrado um tabuleiro final, este é retornado, senão adicionam-se as expansões no início da lista de tabuleiros para expandir. Se a lista de tabuleiros ficar vazia, significa que não foram encontradas soluções e continua-se o algoritmo com o limite de profundidade incrementado.

## Pesquisa ganaciosa

A pesquisa gananciosa está definida no ficheiro *logic/Greedy.java*. Enquanto a lista de tabuleiros por expandir não estiver vazia, expande-se o primeiro elemento e verifica-se se as expansões originam um tabuleiro final. Caso seja encontrado um tabuleiro final, este é retornado, senão adicionam-se as expansões no final da lista de tabuleiros para expandir, seguindo-se a ordenação da lista de acordo com a função de avaliação do tabuleiro que utiliza como heurística o número de blocos entre o bloco vermelho e a saída mais a sua distância à saída. Se a lista de tabuleiros ficar vazia, significa que não foram encontradas soluções.

## Pesquisa utilizando algoritmo A\*

A pesquisa utilizando o algoritmo A\* está definida no ficheiro *logic/AStar.java*. Enquanto a lista de tabuleiros por expandir não estiver vazia, expande-se o primeiro elemento e verifica-se se as expansões originam um tabuleiro final. Caso seja encontrado um tabuleiro final, este é retornado, senão adicionam-se as expansões no final da lista de tabuleiros para expandir, seguindo-se a ordenação da lista de acordo com a soma das funções de avaliação do tabuleiro: número de jogadas que originaram cada tabuleiro e distância ao estado final utilizando como possíveis heurísticas o número de blocos entre o bloco vermelho e a saída, a sua distância à saída ou a soma de ambos estes parâmetros. Se a lista de tabuleiros ficar vazia, significa que não foram encontradas soluções.

## Pesquisa com custo uniforme

Não foi implementada a pesquisa com custo uniforme, visto que o custo do jogo é o número de jogadas, isto é, a profundidade do tabuleiro, pelo que este será igual à pesquisa em largura.

# Experiências e Resultados

Para concluir sobre a eficiência dos algoritmos e ser possível a sua comparação, foram avaliados os tempos de execução, o número de movimentos necessários para resolver o problema e o número de estados visitados por cada algoritmo. Nas seguintes tabelas a heurística utilizada para o algoritmo A\* é a soma da distância do bloco vermelho à saída com o número de blocos a bloquear a saída, que é a mais eficiente como se verá mais à frente.

## Tempo em segundos de execução de algoritmos:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | DFS | BFS | Aprofundamento Progressivo | Ganancioso | A\* |
| Board1 | 0,016 | 0,078 | 0,156 | 0,015 | 0,063 |
| Board2 | 0,047 | 0,249 | 9,148 | 0,063 | 0,141 |
| Board3 | 0,182 | 0,549 | 22,996 | 0,063 | 0,484 |
| Board4 | 0,503 | 3,568 | 2333,361 | 1,000 | 2,528 |
| Board5 | 4,986 | 16,983 | 391,618 | 1,723 | 2,781 |
| Board6 | 0,125 | 0,361 | 1281,231 | 0,14 | 0,361 |

Através desta tabela conclui-se que o algoritmo de pesquisa em profundidade e de pesquisa gananciosa são os de mais rápida execução. Em geral, o algoritmo A\* demora o dobro do tempo destes algoritmos, mas sendo também relativamente rápido. Já a pesquisa em largura demora cerca de quatro vezes mais provando ser menos eficiente em termos de tempo e a pesquisa com aprofundamento progressivo mostra-se bastante mais lenta que as restantes.

A razão pela qual esta última mostra grande discrepância de tempo em relação aos restantes algoritmos deve-se ao facto de neste algoritmo não serem ignorados todos os estados repetidos, apenas aqueles que para além de repetidos, se apresentam a uma maior profundidade do que os mesmos estados que já foram explorados.

## Número de movimentos necessários para alcançar uma solução:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | DFS | BFS | Aprofundamento Progressivo | Ganancioso | A\* |
| Board1 | 83 | 12 | 12 | 12 | 12 |
| Board2 | 347 | 26 | 26 | 28 | 26 |
| Board3 | 1474 | 23 | 23 | 23 | 23 |
| Board4 | 1393 | 47 | 47 | 61 | 47 |
| Board5 | 5812 | 26 | 26 | 28 | 26 |
| Board6 | 1162 | 68 | 68 | 70 | 68 |

A partir da tabela podemos concluir que os algoritmos de pesquisa em largura, aprofundamento progressivo e A\* obtêm resultados ótimos. A pesquisa gananciosa não obtém resultados ótimos, mas não se nota uma diferença muito grande para o número ideal de movimentos atingidos. Já a pesquisa em profundidade apresenta número de movimentos bastante mais elevado.

Através desta tabela e da anterior, pode-se concluir que o algoritmo mais eficiente é o algoritmo de pesquisa informada A\*, que apresenta solução ótima e é relativamente rápido, seguindo-se a pesquisa gananciosa que apesar de não apresentar solução ótima, apresenta uma solução próxima da otimalidade e mais rápida do que a apresentada pelo A\*. É de notar que ambos estes algoritmos são de pesquisa informada.

## Número de estados visitados:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | DFS | BFS | Aprofundamento Progressivo | Ganancioso | A\* |
| Board1 | 103 | 1934 | 4477 | 200 | 1182 |
| Board2 | 448 | 6511 | 132174 | 891 | 3515 |
| Board3 | 2722 | 10623 | 232303 | 887 | 7895 |
| Board4 | 6559 | 24922 | 2220753 | 12780 | 19437 |
| Board5 | 15888 | 43378 | 738207 | 12035 | 18742 |
| Board6 | 1806 | 8001 | 1868530 | 3224 | 7218 |

Através desta tabela conseguimos notar que os algoritmos que percorrem menos estados para chegar a uma solução são o de pesquisa em profundidade e pesquisa gananciosa, o que mostra ser concordante com a tabela relativa ao tempo de execução dos algoritmos.

O algoritmo A\* percorre sempre menos estados do que o de pesquisa em largura, dado que o algoritmo A\* se assemelha ao de largura, menos na ordenação dos estados de acordo não só com a profundidade a que estes estão também com uma heurística adicional.

O algoritmo de pesquisa com aprofundamento progressivo tem que percorrer os mesmos estados várias vezes até chegar à profundidade limitada correspondente à profundidade da solução ótima, pelo que percorre bastantes mais estados do que os restantes algoritmos.

Conclui-se a partir desta tabela que, tendo em conta apenas o número de estados percorridos, a pesquisa gananciosa e a pesquisa em profundidade se mostram os mais eficientes.

## Comparação entre heurísticas utilizadas no algoritmo A\* em relação ao número de estados visitados:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Distância até à saída | Nº de blocos a bloquear a saída | Ambas |
| Board1 | 1632 | 1456 | 1182 |
| Board2 | 5678 | 5951 | 3515 |
| Board3 | 9896 | 9649 | 7895 |
| Board4 | 23225 | 23555 | 19437 |
| Board5 | 29217 | 27703 | 18742 |
| Board6 | 7717 | 7642 | 7218 |

No que toca às heurísticas, foram analisadas três: a distância do bloco vermelho à saída, o número de blocos a bloquear a saída e a soma de ambos esses parâmetros. Todas estas heurísticas mostraram-se como ótimas, na medida em que retornam uma solução no menor número de movimentos possível para todos os tabuleiros iniciais testados.

A heurística da soma da distância do bloco vermelho até à saída com o número de blocos a bloquear a saída mostra-se a mais eficiente, visto que o algoritmo percorre o menor número de estados utilizando-a.

## Comparação entre heurísticas utilizadas no algoritmo A\* em relação ao tempo de execução do algoritmo, em segundos:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Distância até à saída | Nº de blocos a bloquear a saída | Ambas |
| Board1 | 0.271 | 0.166 | 0.119 |
| Board2 | 0.637 | 0.752 | 0.491 |
| Board3 | 2.494 | 1.750 | 1.357 |
| Board4 | 9.506 | 9.779 | 8.201 |
| Board5 | 18.621 | 18.388 | 7.268 |
| Board6 | 0.989 | 0.966 | 0.836 |

Esta tabela mostra ser concordante com a anterior, na medida em que a melhor das heurísticas é a soma entre a distância do bloco vermelho à saída com o número de blocos a bloquear a saída, apresentando um tempo de execução inferior. Deste modo, é possível concluir que essa heurística é a mais eficiente, tendo em consideração ambos os pontos de eficiência considerados.

# Conclusões e Perspetivas de Desenvolvimento

Podemos concluir que o algoritmo mais eficiente é A\*, que apresenta uma solução ótima e eficiente em termos de tempo de execução e número de estados visitados. A heurística mais eficiente a ser usada com este tipo de pesquisa informada é a soma da distância entre o bloco vermelho à saída com o número de bloco a bloquear-lhe a saída. Esta conclusão é concordante com os resultados teóricos esperados.

O único algoritmo que consideramos que não correspondeu às expectativas em termos de eficiência foi o aprofundamento progressivo, visto que neste algoritmo não foi possível proceder à eliminação de estados repetidos do mesmo modo que foi possível em todos os outros algoritmos, tal como foi explicado previamente.

##### Referências Bibliográficas

1. <https://stackoverflow.com/questions/2877724/rush-hour-solving-the-game>