Grupo Nº 54



**Inteligência Artificial**

1.º Semestre 2015/2016

**IA-Tetris**

Relatório de Projecto

Índice

[1 Implementação Tipo Tabuleiro e Funções do problema de Procura 3](#_Toc436330760)

[1.1 Tipo Abstracto de Informação Tabuleiro 3](#_Toc436330761)

[1.2 Implementação de funções do problema de procura 3](#_Toc436330762)

[2 Implementação Algoritmos de Procura 4](#_Toc436330763)

[2.1 Procura-pp 4](#_Toc436330764)

[2.2 Procura-A\* 4](#_Toc436330765)

[2.3 Outros algoritmos 4](#_Toc436330766)

[3 Funções Heurísticas 5](#_Toc436330767)

[3.1 Heurística 1 5](#_Toc436330768)

[3.1.1 Motivação 5](#_Toc436330769)

[3.1.2 Forma de Cálculo 5](#_Toc436330770)

[3.2 Heurística n 5](#_Toc436330771)

[4 Estudo Comparativo 6](#_Toc436330772)

[4.1 Estudo Algoritmos de Procura 6](#_Toc436330773)

[4.1.1 Critérios a analisar 6](#_Toc436330774)

[4.1.2 Testes Efectuados 6](#_Toc436330775)

[4.1.3 Resultados Obtidos 6](#_Toc436330776)

[4.1.4 Comparação dos Resultados Obtidos 6](#_Toc436330777)

[4.2 Estudo funções de custo/heurísticas 6](#_Toc436330778)

[4.2.1 Critérios a analisar 6](#_Toc436330779)

[4.2.2 Testes Efectuados 6](#_Toc436330780)

[4.2.3 Resultados Obtidos 6](#_Toc436330781)

[4.2.4 Comparação dos Resultados Obtidos 7](#_Toc436330782)

[4.3 Escolha da procura-best 7](#_Toc436330783)

# Implementação Tipo Tabuleiro e Funções do problema de Procura

## Tipo Abstracto de Informação Tabuleiro

Tratando-se do jogo *“Tetris”*, neste caso com um tabuleiro 18x10, a maneira mais intuitiva de relacionar as linhas e colunas seria por uma matriz contendo 18 linhas e 10 colunas.

Para tal na primeira entrega usamos um array de dimensões 18 x10, pré preenchido a nil para representar um tabuleiro vazio.

Apesar de funcionar na perfeição, mudamos para uma implementação com *hash tables* na segunda entrega. A razão para a mudança foi em casos que se acede a um valor a partir de um elemento chave, não precisamos de percorrer a lista toda (O(n)) à procura do elemento mas sim apenas ao *key-value* respectivo (O(1)). Isto irá ser benéfico nas procuras por exemplo, podendo estas ser executadas de forma mais rápida e eficiente.

Criamos então o tabuleiro, e como uma *hash table* é um conjunto de pares key/value, vai ser iterada, primeiro pelo número de linhas, depois pelo número de colunas. Guardamos a key como sendo o par linha-coluna, e o respectivo value NIL ou true. Como o tabuleiro inicial está vazio, todos os values são inicializados a NIL.

## Implementação de funções do problema de procura

**Acções**

A nossa função *acções* recebe um estado e devolve a lista de acções válidas a aplicar a este estado, sendo que consideramos que uma acção é válida mesmo que faça o jogador perder e inválida se a colocação da peça for fora dos limites do tabuleiro.

Para podermos implementá-la facilmente, definimos cada peça e cada acção que a esta pode ser aplicada. Criamos uma lista vazia e nesta vamos adicionando acções com as colunas para a peça específica e com a sua configuração. A escolha da orientação está descrita no enunciado – a peça vai rodando 90º no sentido horário. Portanto, a ordem de inserção das configurações na lista é importante.

**Resultado**

A nossa função *resultado* recebe um estado e uma acção, devolvendo um novo estado resultante da aplicação da acção ao estado original. Este novo estado é um novo objecto, não sendo o estado inicial alterado.

Esta função coloca a peça especificada pela acção na posição correcta. Para isso criámos uma função auxiliar, *insere-peca*, que recebe o tabuleiro, peça, linha e uma coluna, que a partir das quais insere a peca no tabuleiro, não devolvendo nada. Verifica-se a existência ou não de colisões, através de uma outra função auxiliar *detecta-colisao*. Se existir colisão, a peça é inserida na linha acima. Se não for detectada colisão e a peça estiver no fundo do tabuleiro, é colocada nessa posição.Depois disto, são actualizadas as listas de peças colocadas e por colocar.

Por último, é feita verificação: se o topo do tabuleiro está ou não preenchido, o que significa o fim do jogo – se estiver não se removem linhas, e devolve-se o estado; se não estiver removem-se as linhas e calculam-se os pontos obtidos, de forma a continuar o jogo, devolvendo também o estado actualizado.

**Solução**

A nossa função *solução* recebe um estado, e devolve true se o estado recebido for solução – ou seja, se o topo não estiver preenchido e não existirem peças por colocar – e nil caso contrário. Para esta função os pontos não são relevantes.

# Implementação Algoritmos de Procura

## Procura-pp

O nosso algoritmo de procura em profundidade primeiro faz a procura em árvore para obter acções que representam soluções para o problema dado.

Como tal, a procura vai explorar uma lista de acções, referentes a um estado inicial. Em cada iteração, identifica devidamente o nó como fronteira ou explorado, passando para o estado seguinte, utilizando uma política LIFO (*last in, first out*), ou seja, o ultimo nó a ser colocado na fronteira é o primeiro que exploramos.

Os nós são listas compostas por três elementos, o problema gerado pela aplicação de uma acção ao problema anterior; a própria acção e o problema anterior. Desta forma, os nós explorados podem ser filtrados de forma a garantir que a lista de acções, devolvida no final, apenas tem as acções que levaram à solução.

Após terminada a procura, caso não seja encontrada nenhuma solução, retorna NIL.

No final, retorna uma lista com todas as acções que reconhece como soluções para o problema especificado, ou retorna NIL quando termina a procura e não encontra soluções.

## Procura-A\*

>Apaga isto aqui>

Para implementarmos a procura A\* utilizámos, para o conjunto de abertos e fechados uma estrutura chamada *node,* com os seguintes campos: problema e acção.

O algoritmo começa por definir o conjunto de nós abertos e fechados, sendo que o conjunto de nós fechados (*closeSet*) começa como uma lista vazia e o conjunto de nós abertos (*openSet*) com o nó gerado a partir do Problema passado como argumento; inicializa os valores de g(x) e h(x) para ser possível calcular o f(x) do primeiro nó; por fim definimos uma hash table que será atualizada com o caminho que a nossa procura toma (*cameFrom*).

Após esta inicialização, o algoritmo começa por retirar o nó com o menor valor de f(x) da lista de abertos e verificar se esse nó é objectivo. Caso não seja, atualizamos lista de abertos e fechados, retirando esse mesmo nó da lista de abertos e colocando-o nos fechados. De seguida, atualizamos a hash table “cameFrom” com o caminho percorrido até esta iteração, geramos os sucessores do nó que estamos a testar, calculamos todos os seus f(x) e colocamos no conjunto aberto. Caso seja objectivo, corremos a função auxiliar *constroi-caminho* que vai juntar todas as ações dos nós navegados, “*cameFrom*”, e retornar essa nova lista revertida.

Caso não consiga encontrar nenhum nó que seja objetivo, então o algoritmos retorna nil.

# Funções Heurísticas

## Heurística Qualidade

### Motivação

Inicialmente tínhamos pensado em utilizar como heurística a função custo-oportunidade, que fornecia já o cálculo de pontos efectivamente conseguidos com uma jogada. Contudo, verificámos que, para um mesmo período de tempo, a heurística custo-oportunidade fazia menos pontos do que a heurística qualidade.

Portanto, a heurística que acabámos por utilizar corresponde a uma função já criada, a função qualidade.   
**Porquê?**

É uma heurística admissível, cujos valores são negativos, pelo que nos permite diminuir em muito o fcost, e assim dar prioridade aos estados com meno  
CONFIRMAR ISTO

Abordar de forma detalhada, *pelo menos*, os seguintes aspectos:

* Ideia inicial, informal, que levou à criação da heurística.
* Informação relevante contida no estado usada no cálculo da heurística.
* Processo de depuração que conduziu à forma final da heurística (outras variantes ensaiadas, etc.)

### Forma de Cálculo

Esta heurística recebe estado e devolve inteiro que corresponde ao valor de pontos ganhos nesse estado até ao momento, em valor negativo:

(\* -1 (estado-pontos estado)).

Descrição *detalhada* da forma final da heurística (incluindo, eventualmente, uma fórmula que permita calcular o seu valor). É *muito importante* que esta descrição seja *concreta*, de modo a que outra pessoa consiga reproduzir a heurística se necessário. Deve também ser *clara* de modo a que qualquer pessoa que tenha lido a motivação da heurística compreenda a sua forma de cálculo.

Esta descrição deve ser complementada com exemplos da sua aplicação, recorrendo a figuras ou outros meios similares que tornem *perceptível* e *imediata* a compreensão da ideia subjacente à heurística.

**.**

**.**

**.**

# Estudo Comparativo

NOTA: Embora nas várias secções seguintes seja pedida a descrição dos testes efectuados, não é necessário repetir essa descrição se os testes já tiverem sido descritos anteriormente, bastando referir onde esta descrição se encontra.

Pretende-se, neste projecto, efectuar quer os algoritmos de procura quer as heurísticas implementadas para o jogo IA-Tetris.

## Estudo Algoritmos de Procura

O primeiro estudo foca-se em comparar as duas implementações pedidas para os algoritmos de procura, a procura em profundidade primeiro e a procura A\*. Caso tenham implementado outro algoritmo de procura (ex. procura local)[[1]](#footnote-1), deverão também compará-lo com as outras duas procuras.

### Critérios a analisar

Descrever quais os critérios a ser usados para comparar os algoritmos e justificar a sua escolha. Alguns exemplos de critérios possíveis: qualidade das jogadas, número máximo de jogadas conseguidas, nós visitados, tempo de execução.

### Testes Efectuados

Descrever os problemas utilizados para efectuar o estudo. Justificar a sua *relevância* para os critérios a comparar.

### Resultados Obtidos

Enumerar os resultados obtidos, sob a forma mais adequada (tabela e gráficos, se necessário).

### Comparação dos Resultados Obtidos

Analisar, caso a caso, os resultados, focando a diferença de desempenho dos vários algoritmos entre si. A análise não deve ser somente uma descrição dos resultados obtidos, mas sim uma análise dos factores que possam ter conduzido a esses resultados.

## Estudo funções de custo/heurísticas

O segundo estudo foca-se em comparar as várias funções de custo e heurísticas implementadas, e perceber qual o custo e heurística com melhores resultados para a qualidade do jogador. **Atenção:** os testes com custos e heurísticas devem ser todos feitos com o mesmo algoritmo de procura.

### Critérios a analisar

Descrever quais os critérios a ser usados para comparar as várias heurísticas e justificar a sua escolha.

### Testes Efectuados

Descrever os problemas utilizados para efectuar o estudo. Justificar a sua *relevância* para os critérios a comparar.

### Resultados Obtidos

Enumerar os resultados obtidos, sob a forma mais adequada (tabela e gráficos, se necessário).

### Comparação dos Resultados Obtidos

Analisar, caso a caso, os resultados, focando a diferença de desempenho dos vários custos e das várias heurísticas entre si. A análise não deve ser somente uma descrição dos resultados obtidos, mas sim uma análise dos factores que possam ter conduzido a esses resultados.

## Escolha da procura-best

Com base nos resultados obtidos nas duas comparações anteriores, devem descrever e justificar qual o algoritmo e respectivas funções custo/heurística para implementar a procura-best criando assim o vosso melhor jogador.

1. Atenção, para que fique claro, não é necessária a implementação de outros algoritmos de procura, mas caso o tenham feito, deverão descrevê-los e compará-los. [↑](#footnote-ref-1)