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

1.1 Tipo Abstracto de Informação Tabuleiro 3

1.2 Implementação de funções do problema de procura 3

2 Implementação Algoritmos de Procura 5

2.1 Procura-pp 5

2.2 Procura-A\* 5

3 Funções Heurísticas 6

3.1 Heurística Custo-Oportunidade 6

3.1.1 Motivação 6

3.1.2 Forma de Cálculo 6

3.2 Heurística Qualidade 6

3.2.1 Motivação 6

3.2.2 Forma de Cálculo 7

# 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\*

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 Custo-Oportunidade

### Motivação

Uma vez que é necessário avaliar a qualidade de cada jogada, precisamos de conseguir medir qual o caminho que nos vai permitir ganhar mais pontos jogando o menor número de peças.

Sabendo isto, se jogarmos muitas peças e tivermos poucos pontos no estado actual, deduzimos que temos tido uma performance baixa. Caso contrário, se a diferença da soma dos valores das peças com a pontuação do estado actual se mantiver ou tiver uma variação pequena, as jogadas realizadas vão de encontro ao objectivo.

**Porquê?**

A heurística mostra ser consistente pois com todos os filhos gerados irão ter um custo superior ou igual ao do pai.

### Forma de Cálculo

Esta heurística recebe um estado e devolve o inteiro que corresponde à diferença entre o máximo possível de pontos ganhos até ao estado actual com os pontos efectivamente conseguidos.

Os pontos efectivamente conseguidos estão guardados na estrutura estado.

Para calcular o máximo de pontos possível, iteramos a lista de peças jogadas no estado actual e guardamos o valor de cada uma.

Por último fazemos a diferença entre o máximo da lista de pontos possível e os pontos efectivamente conseguidos.

## Heurística Qualidade

### Motivação

Inicialmente tínhamos pensado em utilizar como heurística a função custo-oportunidade, que calcula os pontos conseguidos até à data. Contudo verificámos que, para um mesmo período de tempo, a heurística custo-oportunidade não chegava a um estado objectivo. Já a heurística qualidade conseguia obter uma solução.

**Porquê?**

A heurística decresce ao longo do tempo pois a qualidade apenas tem em conta o número de pontos negativos que são acumulados no estado em que estamos. Tendo isso em conta, como escolhemos sempre um nó com o menor valor da função f(x) para ser explorado, esta garante que estamos a percorrer o caminho com a melhor jogada possível.

### Forma de Cálculo

Esta heurística recebe um estado e devolve o inteiro que corresponde ao valor negativo dos pontos ganhos até ao momento.

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