Grupo Nº 64


# Inteligência Artificial

1.º Semestre 2015/2016

# IA-Tetris

## Relatório de Projecto

# Índice

# 1 Implementação tipo Tabuleiro e Funções do problema de Procura

## 1.1 Tipo Abstracto de Informação Tabuleiro

O tipo abstracto de dados Tabuleiro é parte integrante da estrutura Estado que, é sujeita a comparações e cópias constantes por parte dos algoritmos de procura.

Desta forma, para representar Tabuleiro, decidimos criar uma estrutura de dados que desse prioridade à necessidade de efectuar comparações e cópias de forma rápida e eficiente. Com este propósito, optámos por representar o Tabuleiro com uma estrutura composta pelos seguintes campos:

- Um *array* bidimensional de booleanos com dezanove linhas e dez colunas, representativo do "campo de jogo". Correspondendo T a uma casa preenchida e nil a uma casa por preencher. A linha extra revelou-se desnecessária porque se verifica a condição de término de jogo antes da remoção de linhas preenchidas.
- Um vector de inteiros com dimensão igual ao número de colunas que representa o número de casas preenchidas por coluna e permite rápido acesso à "altura das colunas".
- Um vector de inteiros com dimensão igual ao número de linhas para obtenção imediata da informação referente ao número de casas preenchidas em cada linha.
- Um par de inteiros que representam respectivamente, a linha e coluna da última posição preenchida no tabuleiro, ou seja, o índice da linha mais alta e dentro desta a coluna com maior índice preenchida. Esta informação, permite calcular facilmente o deslocamento da origem até à última casa preenchida e optimizar a cópia de tabuleiros.
- Um inteiro que representa o número de casa preenchidas no tabuleiro.

O tabuleiro poderia ser representado apenas pelo primeiro campo, ou qualquer outra combinação de campos que incluísse o primeiro. No entanto, seria necessário percorrer frequentemente o *array* bidimensional para obter informação que assim se encontra disponível através de um simples acesso á cache ou memória. Copiar e comparar tabuleiros seria menos eficiente.

As desvantagens desta estrutura, é o custo acrescido decorrente da actualização de todos os campos adicionais, sempre que há preenchimento de casas ou remoção de linhas no tabuleiro. Além disso, há um ligeiro aumento do espaço ocupado pela estrutura. Este foi o custo que decidimos pagar em troca de um melhor desempenho.

Esta escolha evita na esmagadora maioria dos casos a comparação de todo o campo de jogo, uma vez que, a comparação entre estruturas Tabuleiro é realizada pela ordem inversa da apresentada.

A cópia da Estrutura tabuleiro, apesar de obrigar à reprodução de informação adicional, foi optimizada utilizando a informação dos campos da estrutura, por forma a copiar apenas o estritamente necessário.

## 1.2 Implementação de funções do problema de procura

### Função *accoes (estado)*

Esta função retorna a lista de accões possíveis para um dado estado.

**Algoritmo:**

1. Quando não ha peças por colocar, retorna NIL.

2. Quando o topo está preenchido, retorna NIL.

3. Caso contrário, gera uma lista com as rotações da peça. Para cada rotação, calcula a última coluna onde a peça pode ser colocada (numero de colunas do tabuleiro - largura da peça rodada). As colunas possíveis são todas as que estão entre 0 e esse valor. Tendo a peça e a coluna, cria-se uma estrutura do tipo acção com os dois elementos. Cada acção é então colocada numa lista, retornada pela função.

### Função *resultado (estado, acção)*

Esta função recebe um estado e uma acção. Sem alterar o estado recebido como argumento, aplica-lhe essa accão e retorna o estado resultante.

**Algoritmo:**

Primeiro faz uma cópia do estado recebido como argumento. Depois larga a peça *(ver descrição da função abaixo)* no tabuleiro da cópia. Se houver linhas completas, calcula a nova pontuação, guarda-a no estado resultante e elimina as linhas preenchidas. Depois faz *pop* de uma peça por colocar e *push* da mesma para a lista de peças colocadas. No fim, retorna o estado resultante.

### *Função tabuleiro-larga-peca (tabuleiro, peça, coluna)*

Para as peças mais simples (quadrado, linha), esta função é simples: verifica a altura das colunas onde a peça 'cai', obtém a mais alta de todas e preenche a peça na linha acima.

Para outras peças, em que a base é irregular, é preciso calcular a altura da base. Por exemplo, a **peca-l2** é o L rodado 180º e tem uma base com alturas **(2,0)**. Para estes casos, verifica-se a altura das colunas onde a peça 'cai', subtrai-se a altura da base e obtêm-se as alturas ajustadas à peça. Estas correspondem ao máximo que a peça pode descer. A linha onde a peça é colocada é igual à altura ajustada mais alta - apesar de poderem haver buracos que levam a um ajuste mais baixo para certas colunas, é o limite superior que determina onde a peça encaixa.

**Nota**: apesar das diferentes descrições, todas as peças usam o mesmo algoritmo. As que têm base regular (todas as colunas com altura de base 0) fazem com que o ajuste das alturas seja nulo. Nesses casos, o algoritmo comporta-se de forma semelhante à primeira descrição.

*Função tabuleiro-preenche-peca! (tabuleiro, peça, linha, coluna)*

Esta função é simples: dadas uma linha e uma coluna iniciais, preenche casas do tabuleiro de acordo com a forma da peça. Como a peça é uma matriz, basta percorrê-la num ciclo. Para cada casa preenchida, soma-se-lhe a linha e coluna iniciais para obter a posição do tabuleiro a preencher, através da função *tabuleiro-preenche!*.

# 2 Implementação Algoritmos de Procura

Para as procuras, seguimos os algoritmos do manual da cadeira. Implementámos a função *general-search*, a estrutura *nó* e a função *expande-no*. Com estes elementos, foi fácil fazer diferentes procuras através de diferentes funções de *enqueuing*.

Adicionalmente, implementámos a função *nos->accoes (no-objectivo)***.** Com excepção do nó inicial, cada nó tem um nó-pai e um operador. Então é possível obter a lista de de acções efectuadas desde o nó inicial ate ao nó objectivo, tal como pedido no enunciado.

## 2.1 Procura-pp

**Função *enqueue-front (nós-actuais, nós-expandidos)***

Usando a *general-search*, a procura em profundidade primeiro pode ser implementada com uma função de *enqueue-front*.

Os nós actuais são os nós gerados anteriormente. Os nós expandidos resultam da última expansão de um nó. Os nós actuais são colocados numa lista nova. Enquanto a lista de nós expandidos tem elementos, é retirado o primeiro para ser colocado no início da nova lista. Quando a lista de nós expandidos for vazia, a função retorna a nova lista.

Essa lista é então usada para futuras expansões - o *general-search* expande sempre o primeiro da lista obtida. Assim, controlando a função de *enqueuing*, controla-se também a ordem de expansão de nós.

```
(defun enqueue-front (nos-actuais nos-expandidos)

    (let ((nos-a-adicionar nos-expandidos))
        (loop while (not (null nos-a-adicionar))
        do (push (pop nos-a-adicionar) nos-actuais))
    nos-actuais))
```

*Figura 1 - função enqueue-front*

## 2.2 Procura-A*

A procura-A* chama uma função de procura melhor primeiro - *best-first-search* - que para além do problema e da heurística, recebe como argumento uma função de avaliação, ou seja, o critério para ordenar a fronteira. A procura *best-first-search* por sua vez faz uso da procura genérica - *general-search*.

Usámos para representar a fronteira uma lista de nós ordenada por valor da função de avaliação. No caso da procura-A* a função de *enqueueing* usada chama-se *enqueue-by-value*. Esta função adiciona os nós gerados à lista fazendo imediatamente a ordenação.

```
(defun enqueue-by-value (nos-actuais nos-novos funcao-avaliacao)

    (let* ((todos (nconc (reverse nos-actuais) nos-novos)))
        (stable-sort todos #'<= :key funcao-avaliacao)
        todos))
```

*Figura 2 - função enqueue-by-value*

Manter a lista ordenada pareceu-nos uma melhor solução, pois quando é escolhido outro nó para expansão basta retirar o 1º elemento da lista (valor mais baixo de função de avaliação), ao invés de percorrer toda a lista à procura do nó pretendido.

# 3 Funções Heurísticas

Nesta secção descrevemos as heurísticas usadas para resolver os problemas. Vamos fazer uso do tabuleiro ilustrado na figura abaixo para indicar o valor devolvido por cada função heurística.

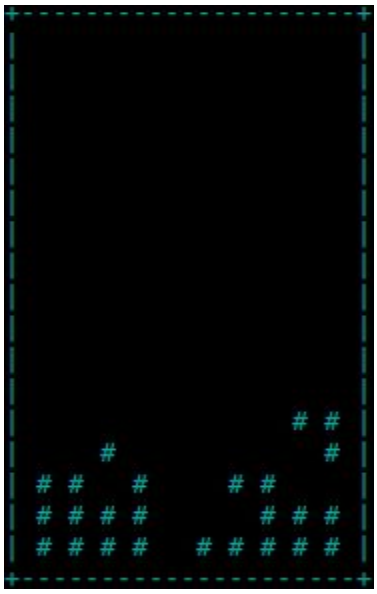

*Figura 3 - Tabuleiro para exemplificar heurísticas*

## 3.1 Heurística 1

A heurística 1 tem por nome *heuristica-dif-colunas* e dado um estado calcula a diferença entre a coluna mais alta e a coluna mais baixa do tabuleiro.

### 3.1.1 Motivação

A ideia inicial era manter o tabuleiro o mais equilibrado possível em termos de altura das colunas. Pretendíamos que as peças não se sobrepusessem umas às outras nas mesmas colunas, estando outras colunas com altura mais baixa, possivelmente mais adequadas para colocar as peças.

No cálculo apenas é usada a estrutura tabuleiro, em particular um par que mantém registo da posição ocupada mais alta - *par-pos-mais-alta* - e um *array* que guarda a altura de cada coluna -

*altura-colunas*.

Ao usar estas estruturas auxiliares a função executa-se em menos tempo.

### 3.1.2 Forma de Cálculo

O cálculo passa por usar a informação que temos da posição ocupada mais alta para achar a altura máxima e o *array* com a altura de cada coluna para encontrar a altura mínima. A fórmula final é

$$resultado = (coluna\ mais\ alta\ - coluna\ mais\ baixa)\ *\ factor$$

, em que factor é um valor que atribui mais peso à heurística, visto que as funções de custo têm valores na ordem das centenas/milhares e resultado é um valor entre zero e dezassete. Neste caso optámos por $factor = 100.$

Para o tabuleiro de exemplo temos então $resultado = (5 - 0) * 100 = 500.$

## 3.2 Heurística 2

A heurística 2 tem por nome *heuristica-casas-ocupadas* e dado um estado retorna o número total de posições preenchidas (casas ocupadas).

### 3.2.1 Motivação

A ideia inicial seria fazer o maior número de linhas possível, traduzindo-se em menos posições preenchidas.

Para o cálculo recorremos a uma variável presente na estrutura tabuleiro - *total-ocupadas* - que mantém exactamente o valor do número de posições preenchidas.

Evoluir para a forma final em que se usa esta variável traduziu-se numa redução significativa do tempo de execução da função heurística, que ao invés de percorrer todo o tabuleiro se limita a fazer um acesso a uma variável.

### 3.2.2 Forma de Cálculo

O resultado desta heurística é simplesmente o valor da variável referida multiplicada por um factor.

$$resultado = total\ casas\ ocupadas * factor$$

Como o valor *total casas ocupadas* pode tomar valores entre 0 (tabuleiro vazio) e 170 (tabuleiro cheio à excepção da 18º linha) optámos por $factor = 25$.

Para o tabuleiro de exemplo, $resultado = 25 * 25 = 625$.

## 3.3 Heurística 3

A heurística 3 tem por nome *heuristica-buracos* e dado um estado retorna uma soma de parcelas que equivalem a contar para cada coluna o número de posições não preenchidas entre a base do tabuleiro e o topo da coluna, posições a que chamámos "buracos". No tabuleiro da figura 3, as 9ª e 10ª colunas têm, respectivamente, 2 "buracos" e 1 "buraco".

### 3.3.1 Motivação

A ideia era que o tabuleiro ficasse com o menor valor possível de "buracos", passíveis de impedir que se completem o máximo de linhas desejado.

Para o cálculo usamos duas estruturas mantidas no tabuleiro: o *array altura-colunas* para obter a altura de cada coluna e a variável *total-ocupadas*.

Mais uma vez o uso destas estruturas adicionais melhorou o tempo de execução da função heurística.

### 3.3.2 Forma de Cálculo

O cálculo é feito subtraindo o número de casas ocupadas ao valor da soma das alturas das colunas. Idealmente estes dois valores seriam iguais, resultando num tabuleiro sem "buracos". A fórmula final é

$$resultado = (soma\ das\ alturas\ das\ colunas - total\ casas\ ocupadas) * factor$$

Como em média num jogo de *Tetris* não é provável que o número de buracos seja elevado optámos por ter $factor = 500$.

Para o tabuleiro de exemplo, $resultado = (30 - 25) * 500 = 2500$.

## 3.4 Heurística 4

A heurística 4 tem por nome *heuristica-altos-e-baixos* e calcula a soma do valor absoluto das diferenças das alturas de colunas adjacentes.

### 3.4.1 Motivação

A ideia inicial era não permitir diferenças grandes entre alturas de colunas adjacentes, como acontece entre os pares de colunas $(1,2)$ e $(2,3)$ no tabuleiro ilustrado abaixo.

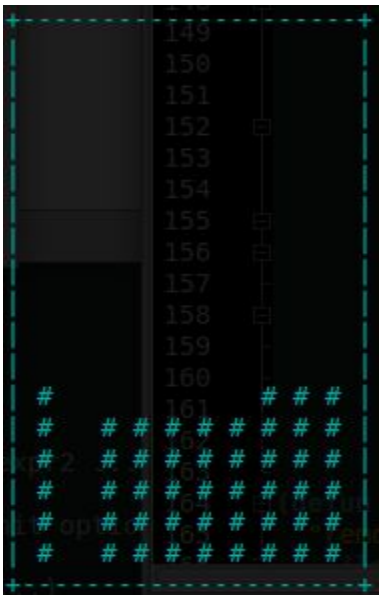

*Figura 4 - Tabuleiro exemplo para a heurística 4*

Para calcular o valor da heurística apenas precisámos do *array altura-colunas* da estrutura tabuleiro para obter o valor da altura para cada coluna.

Mais uma vez utilizar esta estrutura levou a ganhos de performance em termos de tempo de execução.

### 3.4.2 Forma de Cálculo

O resultado é dado pela fórmula

$$resultado = ( |altura_{col1} - altura_{col2}| + ... + |altura_{col9} - altura_{col10}| ) * factor$$

Num caso extremo a soma dos valores absolutos das diferenças entre colunas adjacentes seria de 85 (colunas de altura 17 alternadas com colunas de altura 0). Para obtermos os melhores resultados definimos $factor = 65$ .

Para o tabuleiro de exemplo, $resultado = (0 + 1 + 1 + 3 + 1 + 2 + 0 + 2 + 0) * 65 = 650$ .

## 3.5 Heurística 5

A heurística 5 tem por nome *heuristica-best* e foi a heurística que achámos oferecer melhores resultados nas procuras que efectuámos. Esta é uma combinação linear de três das heurísticas mencionadas: "heuristica-casas-ocupadas", "heuristica-buracos" e "heuristica-altos-e-baixos".

### 3.5.1 Motivação

A ideia desta heurística seria obter os melhores resultados possíveis usando combinações lineares das heurísticas que tiveram uma performance satisfatória quando usadas por si só.

$$resultado = a * heuristica_2 + b * heuristica_3 + c * heuristica_4$$

Os valores finais para os coeficientes foram obtidos por experimentação com vários tabuleiros.

### 3.5.2 Forma de Cálculo

A fórmula final é

$$resultado = 0.5 * heuristica_2 + 0.3 * heuristica_3 + 0.1 * heuristica_4$$

As heurísticas 2 e 3 têm maior peso pois mostraram os melhores resultados individualmente. A heurística 4 ajuda a equilibrar o tabuleiro, mantendo reduzidas as diferenças entre as alturas das colunas.

Para o tabuleiro de exemplo, $resultado = 0.5 * 625 + 0.3 * 2500 + 0.1 * 650 = 1127.5$ .

# 4 Estudo Comparativo

## 4.1 Estudo Algoritmos de Procura

Para a *procura-best* usámos o A*, tirando proveito das heurísticas definidas acima.

### 4.1.1 Critérios a analisar

Os critérios de comparação que usamos são a qualidade das jogadas e o tempo de execução. Consideramo-los os critérios mais relevantes porque analisam a capacidade do algoritmo em conseguir pontuações mais altas em tempo útil.

### 4.1.2 Testes Efectuados

Teste 1 - 3 peças (O I I) num tabuleiro vazio

Teste 2 - 3 peças (O I I) num tabuleiro parcialmente preenchido

Teste 3 - 10 peças (S I Z I T O O L S I) no mesmo tabuleiro inicial

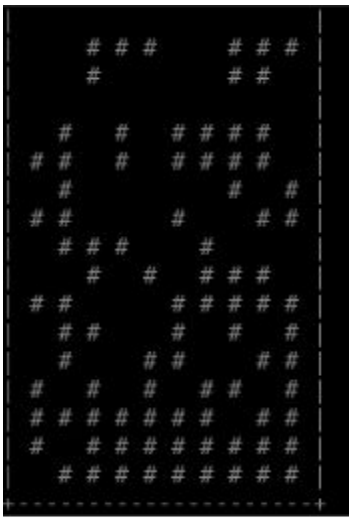

*Figura 4 - Tabuleiro do teste 2 e 3*

### 4.1.3 Resultados Obtidos

Os tempos obtidos foram medidos com a função *time*. A pontuação foi verificada usando a função *desenha-estado*.

Foi usado um computador pessoal, com 4 cores de 2GHz e 6GB de RAM.

| Testes \ Procuras | Procura-pp | Procura-best |
|---|---|---|
| **Teste 1** | 0.22 segundos<br>0 pontos<br>663,912 Bytes | 3.60 segundos<br>100 pontos<br>48,042,600 Bytes |
| **Teste 2** | 0.25 segundos<br>0 pontos<br>647,352 Bytes | 0.35 segundos<br>0 pontos<br>2,412,776 Bytes |
| **Teste 3** | 0.67 segundos<br>200 pontos<br>6,851,072 Bytes | 8.91 segundos<br>400 pontos<br>126,765,104 Bytes |

### 4.1.4 Comparação dos Resultados Obtidos

A procura em profundidade é mais rápida e usa menos memória, mas não é óptima. A procura A* com heurísticas do Tetris permite obter pontuações maiores, mas demora mais tempo e também utiliza mais memória.

A optimalidade é o factor principal da complexidade temporal e espacial para este problema. É relativamente fácil obter soluções não óptimas com a pesquisa em profundidade primeiro - não existem caminhos infinitos, quando uma jogada leva a uma derrota é fácil mudar de caminho. Para atingir optimalidade, a procura A* gera e expande um número de nós muito elevado, usando mais memória e sem conseguir resolver certos problemas em tempo útil.

## 4.2 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.

### 4.2.1 Critérios a analisar

Os critérios de comparação que usamos são a qualidade das jogadas e o tempo de execução. Consideramo-los os critérios mais relevantes porque analisam a capacidade do algoritmo em conseguir pontuações mais altas em tempo útil.

### 4.2.2 Testes Efectuados

Foi usado como referência o teste 25 do Mooshak.

### 4.2.3 Resultados Obtidos

| Heurística<br>Teste | Diferença de colunas | Casas ocupadas | Buracos | Altos e baixos | Best |
|---|---|---|---|---|---|
| **Teste 25 com custo-oportunidade** | 1.43 segundos<br>600 pontos<br>195.66 MB | 0.1 segundos<br>600 pontos<br>2.35 MB | 0.72 segundos<br>600 pontos<br>12.5 MB | 1.23 segundos<br>600 pontos<br>17.3 MB | 0.15 segundos<br>600 pontos<br>3.97 MB |
| **Teste 25 com qualidade** | 0.007 segundos<br>100 pontos<br>0.36 MB | 0.005 segundos<br>400 pontos<br>0.279 MB | 0.005 segundos<br>400 pontos<br>0.279 MB | 0.005 segundos<br>100 pontos<br>0.279 MB | 0.004 segundos<br>400 pontos<br>0.296 MB |

### 4.2.4 Comparação dos Resultados Obtidos

Do quadro podemos observar que usando a função *qualidade* obtemos melhor performance em tempo e espaço, mas as pontuações ficam aquém do óptimo. A procura é mais rápida porque a *qualidade* tem apenas em conta as linhas que foram completadas. Ora, como completamos poucas linhas, o custo vai ser semelhante para vários estados, e o algoritmo chega mais rápido a uma solução (não óptima). Com o *custo-oportunidade* isso já não acontece. Há mais retrocessos mas chega-se à pontuação óptima.

Comparando as heurísticas entre si podemos ver que a heurística *casas-ocupadas* e a *heurística-best* obtêm os melhores resultados; a heurística que tem mais peso na *heurística-best* é exactamente a *heurística-casas-ocupadas*. Neste teste a *heurística-best* acaba por não ser a melhor, mas noutros testes, principalmente com tabuleiros mais desequilibrados em termos de altura das colunas, é normalmente melhor.

## 4.3 Escolha da procura-best

Optámos por usar a procura-A* porque garante melhores pontuações em relação à procura em profundidade primeiro, mesmo que esta última ofereça melhores tempos.

Usámos a *heuristica-best* descrita na secção 3.5. Esta garante na maior parte das vezes a pontuação óptima.

Como função de custo escolhemos a função *custo-oportunidade.* A função *qualidade* não é uma boa medida do custo do caminho até um estado. As procuras em que esta era usada não tinham uma boa performance, tanto em espaço como em tempo.