



# **Inteligência Artificial**

1.º Semestre 2015/2016

## **IA-Tetris**

### **Relatório de Projecto**

Pedro Duarte - 78328

João Almeida - 78451

Gonçalo Fialho - 79112

## Índice

### **1 Implementação Tipo Tabuleiro e Funções do problema de Procura**

- 1.1 Tipo Abstracto de Informação Tabuleiro
- 1.2 Implementação de funções do problema de procura

### **2 Implementação Algoritmos de Procura**

- 2.1 Procura-pp
- 2.2 Procura-A\*
- 2.3 Outros algoritmos

### **3 Funções Heurísticas**

- 3.1 Heurística 1
  - 3.1.1 Motivação
  - 3.1.2 Forma de Cálculo
- 3.2 Heurística n

### **4 Estudo Comparativo**

- 4.1 Estudo Algoritmos de Procura
  - 4.1.1 Critérios a analisar
  - 4.1.2 Testes Efectuados
  - 4.1.3 Resultados Obtidos
  - 4.1.4 Comparação dos Resultados Obtidos
- 4.2 Estudo funções de custo/heurísticas
  - 4.2.1 Critérios a analisar
  - 4.2.2 Testes Efectuados
  - 4.2.3 Resultados Obtidos
  - 4.2.4 Comparação dos Resultados Obtidos
- 4.3 Escolha da procura-best

# 1 Implementação Tipo Tabuleiro e Funções do problema de Procura

## 1.1 Tipo Abstracto de Informação Tabuleiro

O tipo de estrutura de dados utilizada para representar o tipo tabuleiro foi o **array** por ser a representação mais aproximada à realidade do problema. Esta estrutura de dados permite que sejam armazenados elementos e/ou conjuntos de elementos de tal forma que cada um dos elementos possa ser identificado por um índice.

Tal como o problema do Tetris propõe, é necessário que exista uma estrutura bidimensional (linhas e colunas) que indique quais as posições do tabuleiro que estão a ser ocupadas. Para identificar essas zonas são usados os **valores booleanos** do LISP (true e nil).

Esta representação é mais cómoda por ser das representações mais simples para aceder e alterar valores internos da estrutura (posições) na linguagem LISP.

Uma das razões que também nos levou a escolher a estrutura **array** foi devido às funções *tabuleiro->array* e *array->tabuleiro*, que para simplificar a complexidade do código tornou a solução destas funções trivial.

Para a manipulação e acesso da estrutura foram usadas as funções:

- *aref* - para retornar o conteúdo dos elementos do tabuleiro (posição ocupada ou não ocupada);
- *setf* - para manipular o conteúdo dos elementos do tabuleiro (ocupar ou desocupar posição) ;
- *dotimes* - para percorrer toda a estrutura do tabuleiro (percorrer linhas, colunas ou ambos).
- *array-dimension* - para retornar o tamanho do tabuleiro (tamanho das linhas e/ou colunas);

A implementação do tabuleiro também poderia ter sido feita utilizando a estrutura **list** que corresponde a uma lista de elementos, porém as funções de manipulação e acesso para esta estrutura eram a nosso ver, mais complexas e ambíguas que as operações utilizadas para **array's**, tornando assim a sua implementação menos simples.

## 1.2 Implementação de funções do problema de procura

Foram criadas várias funções para o problema de procura do IA-Tetris, entre elas (e as mais relevantes) temos as funções *accoes*, *resultado*, *solucao*, *qualidade* e *custo-oportunidade*.

- *accoes*:

Esta é das funções mais importante em todo o código IA-Tetris, pois é nesta função que são devolvidas todas as jogadas possíveis para o tabuleiro.

A função recebe como argumento um estado que corresponde à situação em que o jogo se encontra num dado momento e pretende devolver uma lista de acções que podem ser realizadas nesse mesmo estado (daí o nome da função *accoes*).

Analisando o algoritmo internamente podemos concluir que é criada uma lista auxiliar, onde serão guardados as acções possíveis para a próxima jogada no tabuleiro. De seguida o programa verifica que peça deve ser colocada no tabuleiro (a primeira na lista **pecas-por-colocar** que pertence à estrutura estado). Irá ser gerado um ciclo para cada rotação da peça e usando uma função auxiliar (*posicao-valida*) é verificado se a peça pode ser colocada em cada uma das colunas do tabuleiro, se a posição for válida (ou seja, a peça pode ser colocada em determinada coluna do tabuleiro) é criada uma **accao** (*cria-accao*) e posteriormente colocada na lista de auxiliar (*lista-accoes* referida anteriormente).

A função auxiliar *posicao-valida* recebe como argumentos um tabuleiro, uma peça, uma linha e coluna, e devolve um **valor lógico** que indica se a peça fornecida nos argumentos pode ser colocada no tabuleiro, na linha e coluna fornecidos. Esta função percorre cada quadricula do tabuleiro e da peça, e utiliza a operação lógica *and* para determinar se a peça sobrepõe alguma das peças que já se encontram no tabuleiro. Se esta condição se verificar significa que a posição onde estamos a querer colocar a peça não é válida (sobreposição da peça com outra que já lá se encontra), e é através desta informação que a função *accoes* conhece se pode ou não colocar a peça em determinada coluna do tabuleiro.

Para além de verificar a sobreposição de peças, a função *posicao-valida* verifica também se as peças podem ser colocadas dentro dos limites do campo, impedindo assim que outras funções mais à frente retornem excepções por tentarem aceder a posições fora do limite do tabuleiro.

- *resultado*:

A função *resultado* devolve o estado resultante de aplicar a acção a um determinado estado, o estado retornado por esta função é utilizado mais à frente para nas funções de procura ser determinado qual o estado a escolher de entre  $n$  estados diferentes, esta escolha é feita consoante o tipo de algoritmo de procura aplicado.

Na função *resultado* o algoritmo começa por colocar a peça dada pela acção no tabuleiro e de seguida calcula o número de linhas preenchidas, de modo a limpar e a incrementar o número de pontos caso se verifique que existam linhas que possam ser limpas.

## 2 Implementação Algoritmos de Procura

### 2.1 Procura-pp

A função *procura-pp* recebe um problema e tenta resolvê-lo utilizando o algoritmo de procura em profundidade primeiro. Esta é uma procura não informada, isto é, não usa qualquer tipo de heurística

para escolher qual o próximo nó a expandir. Este algoritmo consiste em escolher sucessivamente o primeiro nó de uma lista de nós abertos (inicialmente, apenas o estado inicial), verifica se é solução e em caso afirmativo retorna o caminho (lista de acções) desde o estado inicial até a esse estado. Caso este nó não seja solução, são gerados novos nós sucessores a partir da lista de acções do nó actual, e estes são colocados na lista de abertos. Esta procura retorna sem sucesso caso a lista de abertos esteja vazia e não tenha sido encontrada uma solução. Na implementação do nosso projecto, é usada uma função auxiliar reconstruct-path para determinar o caminho desde a solução até ao estado inicial, que é utilizada em todas as funções de procura. Para auxiliar todo o processo da procura, é utilizada uma estrutura “node”, que será explicada na próxima alínea.

## 2.2 Procura-A\*

A função procura-A\* recebe um problema e tenta resolvê-lo utilizando o algoritmo de procura A\*. Esta procura é informada, ou seja, o procedimento de escolher o próximo nó a expandir é feito com recurso a uma função heurística  $f(n) = g(n) + h(n)$ , em que  $h(n)$  é a função que determina o valor da heurística para o nó  $n$ , e  $g(n)$  é a função que determina o custo de caminho desde o nó inicial até ao nó  $n$ .

Para auxiliar este processo, recorreremos ao uso de uma estrutura auxiliar “node” (nó). Esta estrutura contém um estado, o pai desse estado (i.e, o estado que o gerou), a acção que o gerou, o valor da função  $f$  para esse estado e a ordem pelo qual foi gerado (isto é, o estado inicial tem ordem 1, os sucessores do estado inicial tem ordem 2, 3, etc). O valor de  $f$  de um nó é computado assim que o nó é gerado.

Não é feita qualquer ordenação à lista de abertos. Para escolher o melhor nó, é percorrida a lista de abertos à procura do nó com menor valor de  $f$  ou, em caso de empate, aquele que foi gerado à mais tempo (i.e, aquele que tem o maior número de ordem). Este processo é feito na função auxiliar select.

Tal como na procura em profundidade primeiro, esta procura retorna insucesso caso a lista de abertos esteja vazia e não tiver sido encontrada uma solução.

## 2.3 Outros algoritmos

Não foram implementados mais algoritmos além dos pedidos no enunciados, visto que achamos que estes seriam suficientes para conseguir fazer o melhor jogador.

### 3 Funções Heurísticas

#### 3.1 Heurística 1 - *coluna mais alta*

##### 3.1.1 Motivação

Esta heurística tem como ideia base tentar minimizar a altura máxima do tabuleiro, visto que isto minimiza o risco de o jogo ser perdido. Apenas é usado o tabuleiro do estado, necessário para calcular a coluna mais alta e a sua altura.

##### 3.1.2 Forma de Cálculo

Esta heurística, é calculada com o auxílio da função *coluna-mais-alta*, que dado um estado, devolve um inteiro correspondente à coluna com a altura mais alta. O seu procedimento pode-se reduzir à seguinte fórmula:

$$\text{Coluna mais alta} = \max (\text{altura.coluna } i)$$

Isto é computado percorrendo todas as colunas do tabuleiro e guardando a coluna mais alta como resultado, que é depois retornado. De seguida, é calculada a altura desta coluna.

**Note-se** que o procedimento de calcular qual a coluna mais alta (pela função *coluna-mais-alta*) não devolve a *altura* desta e sim *qual* a coluna pois achamos que, para efeitos de escalabilidade, i.e, caso fosse preciso aceder a esta coluna específica no futuro, seria mais prático fazê-lo através desta função, ao invés de ter duas funções semelhantes em que a única diferença seria o valor de retorno.

```

+-----+ Proxima peca:NIL
|
|
+-----+ Pontuacao:NIL
|
|
| # # # # # # # #
| # # # # # # # #
| # # # # # # # #
| # # # # # # # #
+-----+
NIL
[37]> (tabuleiro-altura-coluna (estado-tabuleiro estado) (coluna-mais-alta estado))
4

```

Figura 3.1 - *Coluna mais alta*

## 3.2 Heurística 2 - *bumpiness*

### 3.2.1 Motivação

Esta heurística tem como base a ideia de que quanto mais “plano” o tabuleiro, melhor. Isto é, um tabuleiro plano significa uma maior “regularidade” das peças e controlo das próximas jogadas, o que vai implicar um melhor desempenho e menor risco. Assim, visamos a diminuir a “bumpiness” (ou irregularidade) do tabuleiro. Para isso, dado um estado do jogo, apenas é necessário utilizar o tabuleiro para calcular este valor.

### 3.2.2 Forma de Cálculo

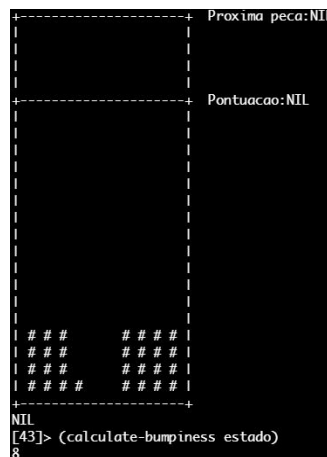
Esta heurística é calculada com auxílio da função `calculate-bumpiness`, que recebe um estado e devolve um inteiro correspondente à irregularidade do tabuleiro correspondente ao estado, e é dada pela seguinte fórmula:

$$\text{Bumpiness} = \sum_{i=0, j=1}^{i=j-1} \text{abs}(\text{altura.coluna } i - \text{altura.coluna } j),$$

onde  $i$  e  $j$  são 2 colunas consecutivas (ex: coluna 0 e coluna 1).

Esta função procede da seguinte forma:

1. Inicializa uma variável resultado
2. Para todas as colunas do tabuleiro
3. Soma ao resultado o valor absoluto da diferença de alturas entre 2 colunas consecutivas.



```

+-----+ Proxima peça:NIL
|       |
|       | Pontuacao:NIL
|       |
|       |
|       |
|       |
|       |
|       |
|       |
|       |
+-----+
| ###  ### |
| ###  ### |
| ###  ### |
| ###  ### |
+-----+
NIL
[43]> (calculate-bumpiness estado)
8

```

Figura 3.2 - *Bumpiness*

## 3.3 Heurística 3 - *altura composta*

### 3.3.1 Motivação

Esta heurística tem uma ideia base semelhante à heurística 1 anteriormente referida, na medida em que visa minimizar a altura do tabuleiro. No entanto, enquanto a heurística 1 tenta minimizar a altura de colunas pontuais, esta visa minimizar a altura composta de todas as colunas, ou altura agregada (*aggregate height*). De igual forma, apenas é necessário o tabuleiro do estado.

### 3.3.2 Forma de Cálculo

Esta heurística é calculada com o auxílio da função *calculate-aggregate-height*, que recebe um estado e devolve a altura composta de todas as colunas do tabuleiro do estado. Esta função pode ser reduzida à seguinte fórmula:

$$\text{Aggregate height} = \sum_i \text{altura.coluna } i$$

Este cálculo é trivial, pois apenas é necessário percorrer todas as colunas do tabuleiro e ir somando as suas alturas (nível da peça mais alta).

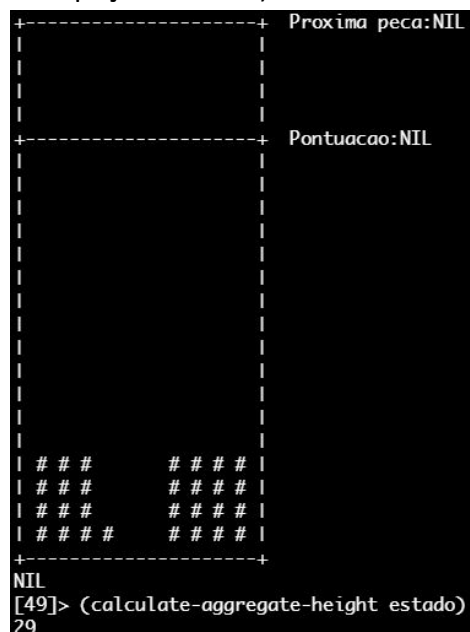


Figura 3.3 - *Aggregate height*

## 3.4 Heurística 4 - *bolhas*

### 3.4.1 Motivação

Esta heurística parte do mesmo princípio da heurística número 2, pois também visa a reduzir a



irregularidade do tabuleiro. No entanto, enquanto a heurística 2 é utilizada para reduzir a irregularidade das alturas das colunas do tabuleiro, esta pretende diminuir o número de bolhas (ou buracos) no tabuleiro.

Esta heurística é muito importante, pois um buraco no tabuleiro é algo que é difícil de eliminar, visto que seria necessário primeiro limpar todas as linhas que se encontram em cima da linha do buraco para o poder preencher e só assim limpá-lo.

Tal como as heurísticas anteriormente apresentadas, esta também necessita apenas do tabuleiro do estado para poder ser computada.

### 3.4.2 Forma de Cálculo

Esta heurística é calculada com o auxílio da função `calculate-bubbles`, que recebe um estado e devolve o número de bolhas (ou buracos) do tabuleiro correspondente ao estado. Esta função tem o seguinte procedimento:

1. Para cada coluna (percorridas da esquerda para a direita, embora irrelevante)
  - 1.1. Para cada linha (percorridas de baixo para cima), até à altura máxima dessa coluna:
    - 1.1.1. Se houver alguma posição não preenchida, incrementa o número de bolhas
2. Retorna o número de bolhas

**Note-se** que este procedimento não incrementa o número de bolhas no limite da altura máxima de uma coluna, pois não para antes de lá chegar:

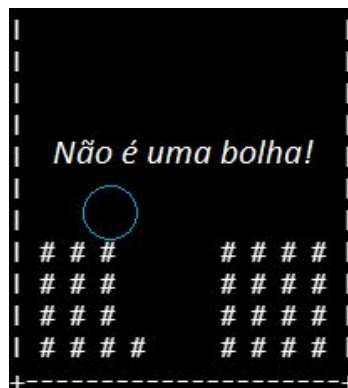


Figura 3.4.1 - Exemplo

Este procedimento pode ser dado pela seguinte fórmula simplificada:

$$\text{Bubbles} = \sum_i^{\text{\#colunas}} \sum_j^{\text{altura}} \text{buraco?}$$

onde buraco? tem o valor 1 se a posição (i,j) não estiver preenchida e 0 caso contrário (tendo em atenção a nota referida acima).

```

+-----+ Proxima peça:NIL
|
|
+-----+ Pontuacao:NIL
|
|
|
|
|
|
|
|
| # # # # # # # #
| # # # # # # # #
| # # # # # # # #
| # # # # # # # #
+-----+
NIL
[54]> (calculate-bubbles estado)
0

```

Figura 3.4.2 - Bubbles

### 3.5 Heurística 5 - pontos

#### 3.5.1 Motivação

Esta heurística tem como motivação conseguir **maximizar** o número de pontos, minimizando o negativo do número de pontos de um estado. Ao contrário de todas as outras heurísticas apresentadas, esta heurística necessita apenas dos pontos de um estado e não necessita de qualquer função auxiliar **adicional** (apenas a que foi pedida no enunciado, estado-pontos).

Na prática, esta heurística corresponde à função **qualidade**, que nós achamos importante utilizar como uma heurística, e não como custo de caminho.

#### 3.5.2 Forma de Cálculo

Esta heurística é dada, sem qualquer cálculo, pelo selector estado-pontos que, dado um estado, devolve o número de pontos conseguidos até ao momento. No momento do cálculo de todas as heurísticas (i.e, na sua composição), é usado o valor negativo.

Este valor pode ser obtido trivialmente pela fórmula:

$$\text{Pontos} = \text{pontos.estado}$$

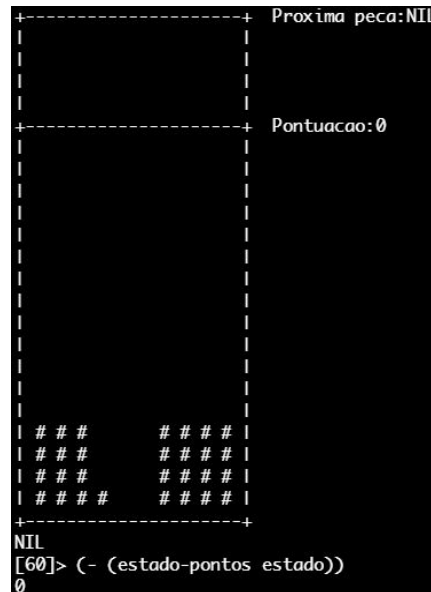


Figura 3.5 - Pontos

## 4 Estudo Comparativo

### 4.1 Estudo Algoritmos de Procura

#### 4.1.1 Critérios a analisar

Nestes testes foram analisados o tempo de execução, a memória usada, o número de nós visitados, os pontos e o número de jogadas conseguidas.

O tempo de execução é crucial para qualquer algoritmo de procura, e embora não seja muito importante minimizá-lo o máximo possível, é importante que a procura seja feita em tempo útil. O mesmo se passa com a memória, que embora menos importante que a velocidade pois hoje em dia é uma coisa que abunda na maior parte dos computadores, não deve ser negligenciada e portanto é claro que a sua utilização deve ser minimizada. No entanto, consideramos o consumo de memória elevada em troco de maior rapidez de execução uma boa troca.

O critério do número de nós visitados é também relevante para tirar conclusões sobre a evolução da procura.

O número de jogadas conseguidas e os pontos permitem observar a qualidade propriamente dita da procura para o problema do tetris, pois de nada serve a procura ser rápida se não consegue fazer jogadas aceitáveis.

#### 4.1.2 Testes Efectuados

Foram utilizados os seguintes problemas: o problema 1, com o tabuleiro inicial parcialmente preenchido, onde foi variado o número de peças por colocar, e o problema 2, com o tabuleiro inicialmente vazio, com a mesma variação do número de peças.

**Note-se** que, para efeitos de simplificação, não foi feita qualquer discriminação entre o tipo de peças, pois achamos que apenas o número de peças seria informação suficiente para obter resultados fiéis. No entanto, em ambos os testes, *ambas as procuras utilizaram as mesmas peças* (inicialmente a peça 'O', depois adicionamos peças consecutivamente até o último teste utilizar 5 peças: O, J, L, T e I).

Ambos os problemas testam fielmente todos os critérios escolhidos e permitem comparar e perceber qual a diferença entre um tabuleiro vazio e um tabuleiro já parcialmente preenchido.

Os testes são realizados no mesmo ambiente de desenvolvimento do projecto, e consistem em escrever as informações relevantes para o terminal, tais como o tempo (com o auxílio da função `time`), o número de nós visitados, etc.

## 4.1.3 Resultados Obtidos

Procura PP						
	# Peças	# Nós visitados	Pontos	# Jogadas	T Exec [s]	Memória [bytes]
<b>Problema 1</b>	1	2	0	1	0.008	29 472
Tabuleiro	2	3	0	2	0.04	122 880
vazio	3	4	0	3	0.08	216 944
	4	5	0	4	0.108	311 872
	5	6	0	5	0.112	360 512
	# Peças	# Nós visitados	Pontos	# Jogadas	T Exec [s]	Memória [bytes]
<b>Problema 2</b>	1	2	0	1	0.012	29 152
Tabuleiro	2	3	0	2	0.044	121 216
parcialmente	3	4	0	3	0.076	213 840
preenchido	4	5	0	4	0.108	307 328
	5	6	0	5	0.109	355 200

Tabela 4.1.1 - Resultados obtidos pela procura PP

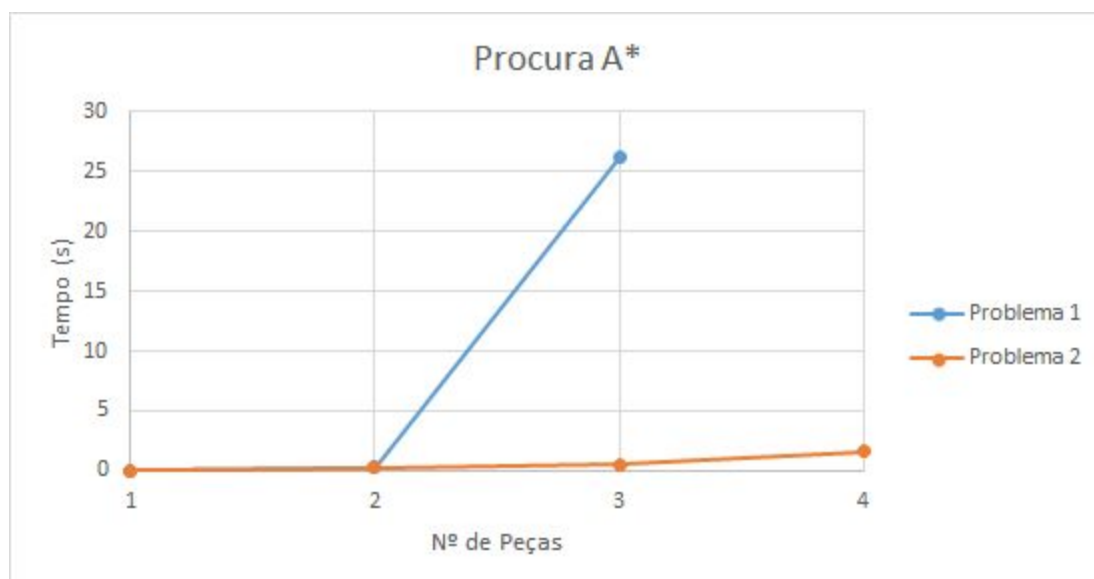


Gráfico 4.1.1 - Tempos obtidos pela procura PP

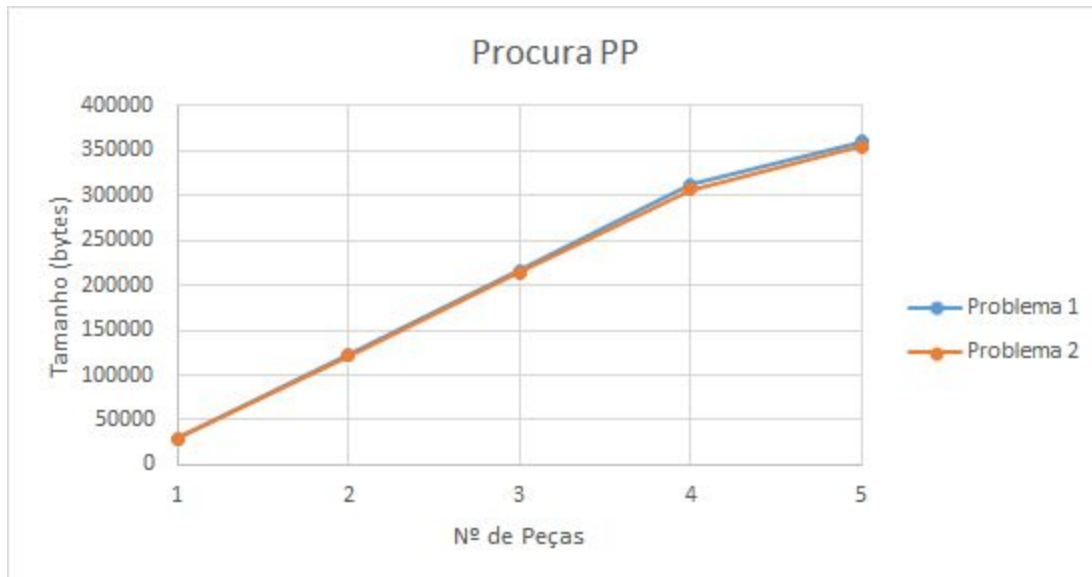


Gráfico 4.1.2 - Memória utilizada pela procura PP

As seguintes procuras A\* foram feitas com  $g$  = custo de caminho e  $h$  = qualidade.

Procura A* ( $g$ = custo de oportunidade e $h$ = qualidade)						
	# Peças	# Nós visitados	Pontos	# Jogadas	T Exec [s]	Memória [bytes]
<b>Problema 1</b>	<b>1</b>	2	0	1	0.012	31 200
Tabuleiro	2	11	0	2	0.296	106 880
vazio	3	317	0	3	26.18	213 987 984
	4	-	-	-	-	-
	5	-	-	-	-	-
	# Peças	# Nós visitados	Pontos	# Jogadas	T Exec [s]	Memória [bytes]
<b>Problema 2</b>	<b>1</b>	2	100	1	0.012	30 688
Tabuleiro	2	8	300	2	0.296	661 568
parcialmente	3	15	400	3	0.496	1 526 560
preenchido	4	38	500	4	1.568	5 308 192
	5	-	-	-	-	-

Tabela 4.1.2 - Resultados obtidos pela procura A\*

**Nota:** alguns resultados não são apresentados nestas tabelas, bem como nas próximas, pois o tempo de execução foi muito superior ao tempo útil e exaustou os recursos limitados do ambiente de execução. No entanto, é claro inferir que os resultados revelariam esta procura praticamente inútil para um número elevado de peças (4 ou 5 para cima).

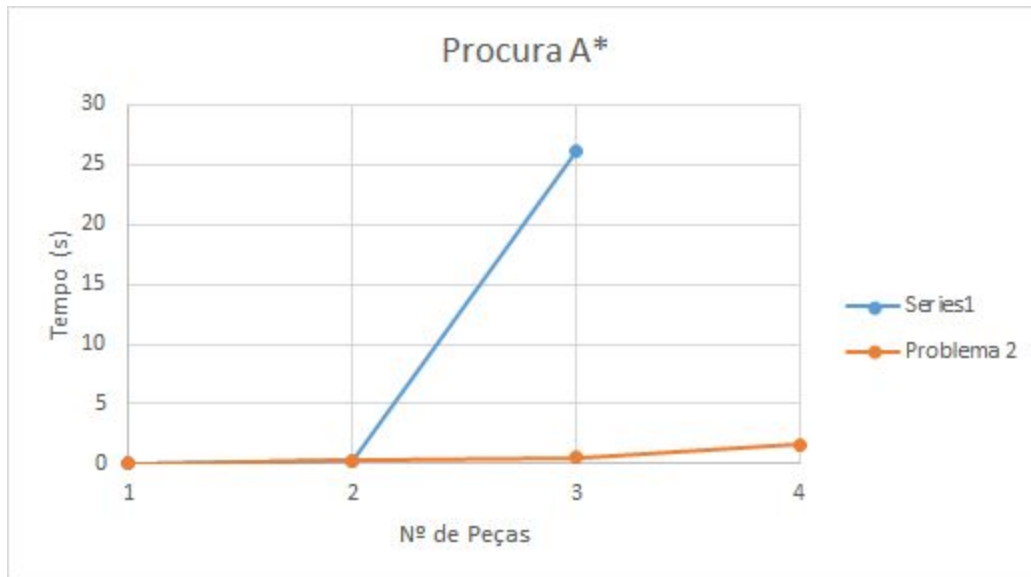


Gráfico 4.1.3 - Tempos obtidos pela procura A\*

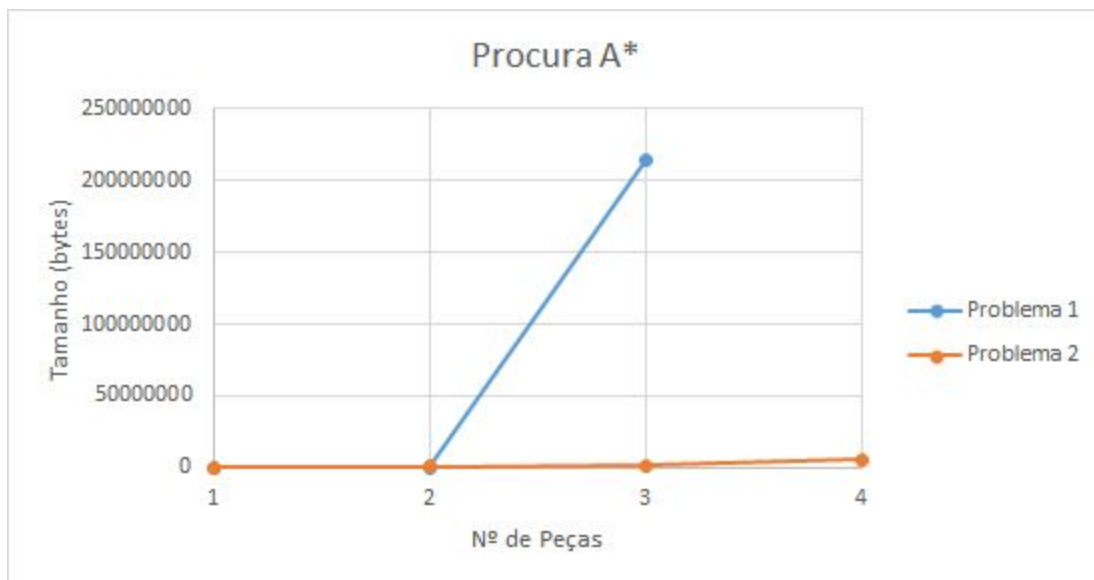


Gráfico 4.1.4 - Memória utilizada pela procura A\*

#### 4.1.4 Comparação dos Resultados Obtidos

Na procura em profundidade primeiro, para todo o número de peças, os resultados dos dois problemas são praticamente semelhantes, tanto para o número de nós visitados como tempo de execução e para a utilização de memória. Este resultado está de acordo com o que esperávamos, dado que na procura em profundidade é sempre escolhido o nó de maior nível na árvore, e o facto de o tabuleiro estar vazio ou parcialmente preenchido não afecta em nada o tempo de execução desta procura.

Quanto ao número de pontos, é normal esta procura ser muito pouco óptima, visto que apenas está focada em encontrar a solução o mais rápido possível. De facto, seria surpreendente observar resultados em que o número de pontos não tivesse sido zero. No entanto, e apesar de não ter sido o caso, com o tabuleiro parcialmente preenchido a probabilidade de obter pontos aumenta, visto que há mais hipóteses de, quando uma peça for colocada no tabuleiro, limpar uma ou mais linhas das que já lá estavam.

Já na procura A\* verifica-se que os resultados dos dois problemas diferem bastante entre si.

Em primeiro lugar, analisemos a diferença entre o número de nós visitados e o número de pontos dos dois problemas. O número de nós visitados com um tabuleiro inicial vazio é maior do que com um tabuleiro inicial parcialmente preenchido, mas verifica-se o contrário para os pontos obtidos. A explicação para isto é bastante simples: a procura A\* foca-se em encontrar a melhor solução possível (neste caso, o maior número de pontos possível) de acordo com a heurística usada, o que significa que a procura não para enquanto não achar esta solução. Como já foi referido acima, num tabuleiro parcialmente preenchido é mais fácil obter pontos do que com um tabuleiro vazio. Assim, para um tabuleiro parcialmente preenchido, é normal que o número de nós visitados seja *menor*, pois haverá mais nós que façam pontos e, por conseguinte, a procura não necessita de procurar tanto pela solução óptima. Consequentemente, o número de pontos obtidos também será maior.

Quanto ao tempo de execução e à memória, verifica-se o mesmo referido anteriormente: quantos mais nós explorados, maior o tempo de execução e maior a memória utilizada.

Comparando as duas procuras, e observando apenas o caso do tabuleiro parcialmente preenchido para efeitos de simplificação, podemos observar que a procura em profundidade encontra solução em tempo útil, enquanto que a procura A\* o tempo de execução é muito elevado a partir de quatro peças, o que demonstra a complexidade exponencial desta procura. No entanto, a procura em profundidade não consegue fazer pontos nenhuns, ao contrário da A\* que devolve sempre o maior número de pontos possível. Isto resulta das propriedades destas procuras, anteriormente referidas: visto que a A\* é óptima em condições específicas, é normal que consiga tantos pontos, e que demore muito mais tempo que a procura em profundidade, visto que o cálculo do valor de  $f$  pode não ser computacionalmente trivial (tanto a função do custo de caminho como a função heurística precisam de ser calculadas para cada nó gerado, pois é isto que garante a optimalidade da solução sob certas condições). Isto também se aplica à memória utilizada pelas procuras: quantos mais nós visitados e expandidos, maior a utilização da memória.



## 4.2 Estudo funções de custo/heurísticas

### 4.2.1 Critérios a analisar

Para comparar as heurísticas vão ser utilizados os mesmos critérios do ponto anterior: o tempo de execução, a memória usada, o número de nós visitados, os pontos e o número de jogadas conseguidas. Embora estejamos a comparar coisas diferentes, achamos que estes critérios continuam relevantes para comparar as heurísticas, pelas mesmas razões acima explicitadas.

### 4.2.2 Testes Efectuados

Os testes foram realizados com a nossa procura-best e com um tabuleiro parcialmente preenchido. A função de custo de caminho usada foi a custo-oportunidade.

Foram testadas as seguintes heurísticas (compostas por heurísticas já referidas):

- $H1 = h4$  (*bubbles*)
- $H2 = h2 + h3 + h4$  (*bumpiness + aggregate height + bubbles*)
- $H3 = 30 \cdot h1 + 5 \cdot h2 + 10 \cdot h3 + 25 \cdot h4 - 0.6 \cdot h5$  (*coluna mais alta + bumpiness + aggregate height + bubbles - pontos*)

**Note-se** que, tal como nos testes feitos anteriormente, não foi feita qualquer discriminação entre o tipo de peças, e serão usadas consecutivamente as peças O, J, L, T e I.

Os testes são realizados no mesmo ambiente de desenvolvimento do projecto, e consistem em escrever as informações relevantes para o terminal, tais como o tempo (com o auxílio da função `time`), o número de nós visitados, etc.

## 4.2.3 Resultados Obtidos

Procura Best						
H1	# Peças	# Nós visitados	Pontos	# Jogadas	T Exec [s]	Memória [bytes]
	1	2	100	1	0.06	55 712
	2	8	200	2	1.168	1 186 416
	3	131	400	3	25.184	37 341 872
	4	150	500	4	28.208	45 076 880
	5	-	-	-	-	-
H2	# Peças	# Nós visitados	Pontos	# Jogadas	T Exec [s]	Memória [bytes]
	1	2	100	1	0.064	54 880
	2	6	300	2	0.776	792 144
	3	30	500	3	5.556	5 627 552
	4	99	500	4	19.144	23 567 008
	5	-	-	-	-	-
H3	# Peças	# Nós visitados	Pontos	# Jogadas	T Exec [s]	Memória [bytes]
	1	2	100	1	0.048	53 192
	2	3	200	2	0.236	237 192
	3	8	400	3	1.244	1 178 520
	4	17	400	4	3.06	2 972 760
	5	684	600	5	89.264	261 055 704

Tabela 4.2.1 - Resultados da procura best

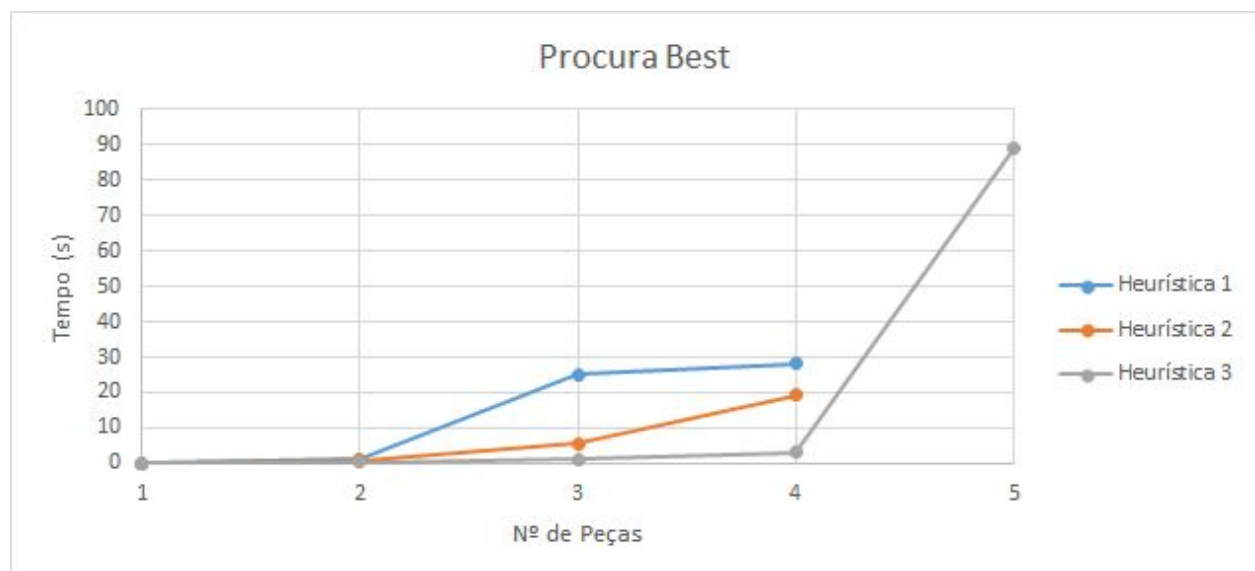


Gráfico 4.2.1 - Tempos obtidos pela procura best

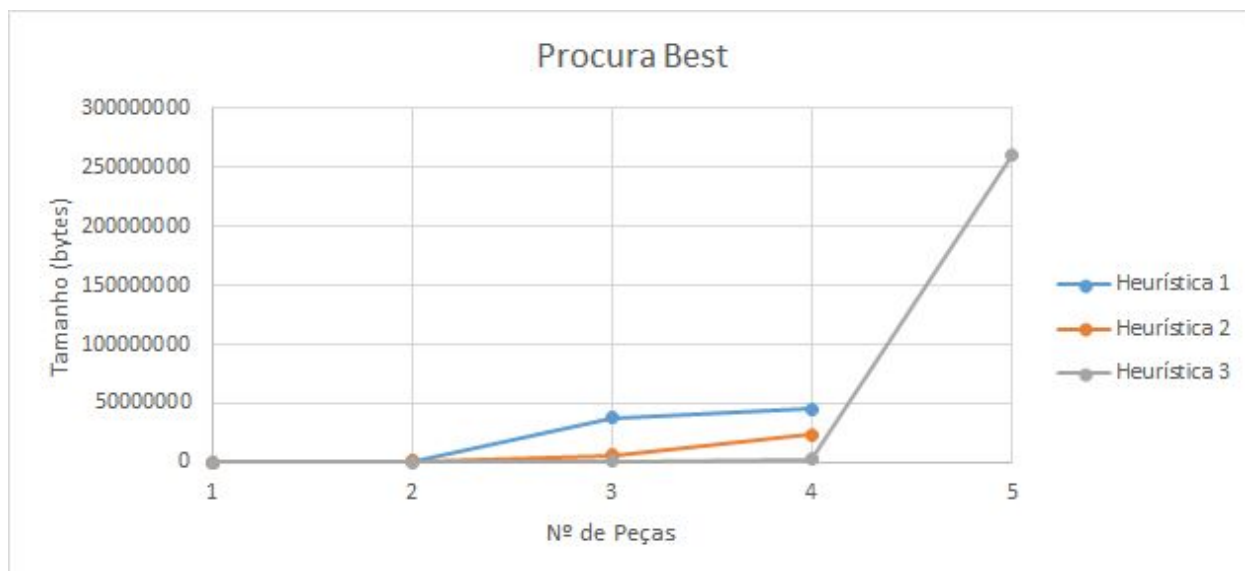


Gráfico 4.2.2 - Memória utilizada pela procura best

#### 4.2.4 Comparação dos Resultados Obtidos

A nossa procura-best (que será explicada com maior pormenor mais à frente) é uma versão da procura A\* com algumas variações na escolha dos nós.

Analiseemos as duas primeiras heurísticas, H1 e H2. A heurística H2 visita menos nós que a heurística H1, isto porque a composição de várias heurísticas (H2) em vez de apenas uma (H1) permite à procura ser mais “directa” ao tentar encontrar uma solução, isto é, quanto mais profundo o estado actual, mais “correcto” está o caminho. Isto faz que, quando a procura precise de recuar, não o precise de fazer muitas vezes. Como há uma selecção dos nós que vão ser vistos, o número de nós que vão ser considerados será também reduzido comparativamente à procura A\* (isto será explicado com mais pormenor mais à frente). Visto que há uma correlação entre o número de nós visitados e o tempo de execução da procura, é normal o tempo de execução ser maior na heurística H1.

Assim, desenvolvemos a heurística final H3, que é composta pelas 5 heurísticas anteriormente apresentadas e tira maior partido da ideia exposta acima (em geral, quanto mais heurísticas melhor). Esta ideia funciona muito bem, pois nenhuma das componentes da heurística é excepcionalmente cara em termos de tempo de computação e memória, o que faz com que haja um bom compromisso entre optimalidade e tempo de computação. De facto, como se pode observar na tabela 4.2.1, não só foi possível obter uma solução para o problema em tempo útil, como se obteve uma pontuação muito boa (note-se que o tabuleiro usado é o mesmo que foi usado nos testes das procuras A\* e procura PP, bem como as peças). Repare-se que, para menos de 5 peças, tanto o tempo de execução como a utilização de memória é **muito** inferior comparativamente à procura best utilizando as outras duas heurísticas H1 e H2. Isto reforça a ideia que uma boa heurística pode aumentar consideravelmente o desempenho de uma procura.

### 4.3 Escolha da procura-best

O nosso algoritmo de procura-best é uma tentativa de melhorar os tempos de execução e a utilização da memória da procura  $A^*$ , onde utilizamos a função de custo de caminho custo-oportunidade e a seguinte heurística, que foi decidida com base nos resultados obtidos pelos testes feitos acima:

$$H = 30*h1 + 5*h2 + 10*h3 + 25*h4 - 0.6*h5$$

Na base do nosso código decidimos calcular a média das funções heurísticas, procurando seleccionar os nós cujas heurísticas que cumpriam um valor acima da média total, e escolhendo apenas estas para fazerem parte da árvore de procura. Deste modo apenas as heurísticas acima da média seriam seleccionadas, o que nos leva a uma procura mais “decidida” e por conseguinte a uma solução mais rápida, visto que vários nós são descartados neste processo, que está baseado no algoritmo genético<sup>1</sup>. Isto significa que irão ser excluídos nós que, apesar de apresentarem uma solução válida para o problema, não serão bons intermediários para uma solução óptima do nosso IA-Tetris.

As escolhas das constantes heurísticas foram decididas empiricamente, onde tentámos ao máximo tornar a heurística o mais optimizada possível de modo a que a nossa procura-best apresentasse sempre os melhores resultados sem que isso implicasse um aumento exponencial de recursos temporais e espaciais.

---

<sup>1</sup> Implementados como simulações de computador em que uma população é seleccionada em busca de soluções melhores, geralmente iniciada a partir de um conjunto de soluções aleatório e evolui por mutação para combinar uma nova população.