

Grupo Nº 48



Inteligência Artificial

1.º Semestre 2015/2016

IA-Tetris

Relatório de Projecto

Índice

1	Implementação Tipo Tabuleiro e Funções do problema de Procura.....	4
1.1	Tipo Abstracto de Informação Tabuleiro	4
1.2	Implementação de funções do problema de procura.....	4
1.2.1	Função <i>accoes</i>	4
1.2.2	Função <i>resultado</i>	4
2	Implementação Algoritmos de Procura	6
2.1	<i>Procura-pp</i>	6
2.2	<i>Procura-A*</i>	6
3	Funções Heurísticas e de Custo de Caminho.....	8
3.1	Função <i>heuristicapontuacao-1</i>	8
3.1.1	Motivação	8
3.1.2	Forma de Cálculo	8
3.2	Função <i>heuristicaltura</i>	9
3.2.1	Motivação	9
3.2.2	Forma de Cálculo	9
3.3	Função <i>custo-altura-agregada</i>	9
3.3.1	Motivação	9
3.3.2	Forma de Cálculo	9
3.4	Função <i>custo-buracos</i>	9
3.4.1	Motivação	9
3.4.2	Forma de Cálculo	10
3.5	Função <i>custo-peças-colocadas</i>	10
3.5.1	Motivação	10
3.5.2	Forma de Cálculo	10
3.6	Função <i>custorelevo</i>	10
3.6.1	Motivação	10
3.6.2	Forma de Cálculo	10
4	Estudo Comparativo	11
4.1	Estudo Algoritmos de Procura	11
4.1.1	Critérios a analisar	11
4.1.2	Testes Efectuados.....	11
4.1.3	Resultados Obtidos.....	12

4.1.4	Comparação dos Resultados Obtidos	13
4.2	Estudo funções de custo/heurísticas	13
4.2.1	Critérios a analisar	13
4.2.2	Testes Efectuados.....	13
4.2.3	Resultados Obtidos.....	14
4.2.4	Comparação dos Resultados Obtidos	14
4.3	Escolha da função <i>procura-best</i>	15
4.3.1	Motivação.....	15
4.3.2	Implementação.....	15
4.4	Implementação do Algoritmo Genético.....	16
4.4.1	População	16
4.4.2	Função de <i>Fitness</i>	16
4.4.3	Selecção.....	16
4.4.4	Reprodução	17
4.4.5	Mutação.....	17

1 Implementação do Tipo Tabuleiro e Funções do problema de Procura

1.1 Tipo Abstracto de Informação Tabuleiro

O tipo tabuleiro é representado internamente por um *array* 2D, de dimensões 18×10 , para se ter acesso $O(1)$ a cada posição do tabuleiro. Esta representação tem também o conveniente de tornar relativamente simples a implementação das várias funções pedidas no enunciado. Em particular, as funções de conversão *array->tabuleiro* e *tabuleiro->array* tornam-se triviais de implementar.

1.2 Implementação de funções do problema de procura

No problema de procura usamos essencialmente as funções: *accoes* e *resultado*, descritas de seguida. Estas são as funções utilizadas pelos algoritmos de procura para efectuar a expansão dos nós das árvores de procura.

1.2.1 Função *accoes*

Dado um estado, esta função devolve uma lista de todas as acções possíveis de se executar com a próxima peça a ser colocada.

De acordo com o enunciado, uma acção é considerada válida se for fisicamente possível coloca-la dentro dos limites laterais do jogo. Assim, dada uma rotação, de largura l , de uma peça, temos $18 - l + 1 = 19 - l$ colunas possíveis onde a colocar (colunas 0 a $18 - l$).

Seguindo este raciocínio, a função *accoes-validas* (definida dentro da função *accoes*) devolve uma lista de acções válidas dada uma rotação particular de uma peça.

A função *accoes* é então trivialmente implementada (1) detectando qual a próxima peça a ser colocada e (2) mapeando a função *accoes-validas* sobre uma lista de todas as rotações dessa peça.

Para terminar, é de notar que a função *accoes-validas* podia ser calculada de forma mais eficiente. Na sua versão actual, cada nova acção criada vai sendo colocada no final da lista de acções, em vez de no início. Ora, isto corresponde à diferença entre computar a lista em tempo quadrático ou em tempo linear, respectivamente. No segundo caso, a ordem correcta da lista pode ser obtida percorrendo o tabuleiro da direita para a esquerda, ou então simplesmente invertendo-a no final. Ambas as soluções teriam complexidade linear.

1.2.2 Função *resultado*

Esta função recebe um estado e uma acção, e devolve um novo estado que resulta de aplicar a mesma sobre o estado original. A função cria um estado inteiramente novo, em vez de alterar o estado original.

A função *resultado* utiliza duas outras funções auxiliares: *tabuleiro-executa-accao!* e *tabuleiro-remove-linhas-completas!*. A primeira é utilizada para, efectivamente, aplicar a acção sobre o tabuleiro. Se após se executar a acção o novo tabuleiro não tiver o seu topo preenchido aplica-se a segunda função. Esta última, para além de remover as linhas preenchidas, devolve o número de linhas removidas, valor esse que é utilizado para determinar a pontuação obtida com a jogada efectuada.

1.2.2.1 Função *tabuleiro-executa-accao!*

A função *tabuleiro-executa-accao!* recebe um tabuleiro e uma acção, e aplica essa acção a esse tabuleiro, produzindo um novo.

Para determinar a linha onde posicionar a peça começa-se por determinar que colunas são abrangidas pela mesma. Por exemplo, uma peça com largura 3 a ser colocada na coluna 0 abrange as colunas 0, 1 e 2. Dada uma lista com as alturas dessas colunas, a linha pretendida é simplesmente o máximo das diferenças entre essas e a altura da primeira posição preenchida nas colunas correspondentes da peça. Por exemplo, imaginemos que queremos colocar uma peça T “em pé” (*peca-t2*) de tal forma que esta abrange colunas com alturas 4, 3 e 2, respectivamente, da esquerda para a direita. A lista dos números de buracos da peça é 1, 0, 1 e, portanto, a lista das diferenças é $4 - 1 = 3$, $3 - 0 = 3$ e $2 - 1 = 1$. A linha pretendida é o máximo das diferenças que, neste caso, é 3.

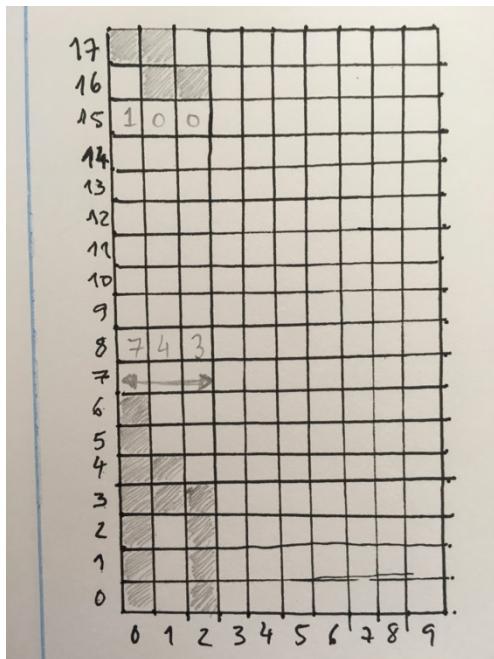


Figura 1-1 – Caso de jogo do Tetris. A peça a ser colocada (na coluna 0) tem largura três e, por isso, abrange as colunas 0, 1 e 2. As alturas das primeiras posições preenchidas das suas colunas são, da esquerda para a direita, 1, 0 e 0. As alturas das colunas abrangidas são 7, 4 e 3.

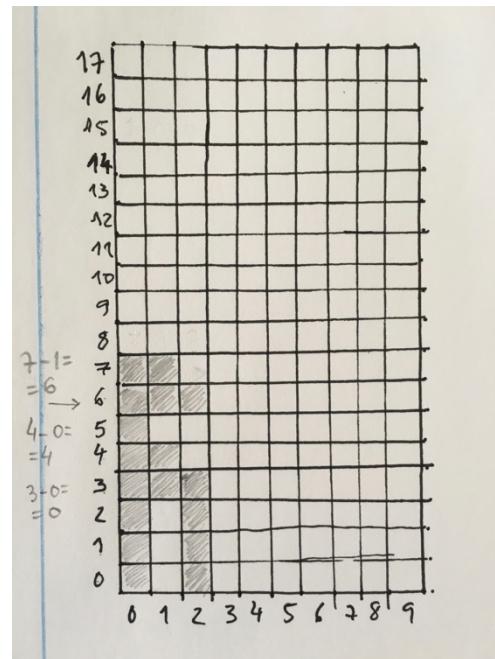


Figura 1-2 – É calculado o máximo das diferenças entre as alturas das colunas e as primeiras posições preenchidas nas colunas da peça. Neste caso, o máximo é $7 - 1 = 6$ e, portanto, a peça será colocada na posição (6, 0).

2 Implementação dos Algoritmos de Procura

2.1 Procura-pp

A função *procura-pp* recebe uma estrutura do tipo *problema* e aplica-lhe uma procura em profundidade primeiro para encontrar uma solução. O resultado é uma lista das acções (um caminho) que são necessárias aplicar ao estado original para chegar ao estado solução.

A procura é caracterizada por, por ordem LIFO dos estados da fronteira e de forma incremental¹, expandir o estado actual e verificar se os estados gerados são solução logo na geração. Se um estado for solução devolve uma lista com a acção que o originou; se não, começa uma procura (recursivamente) a partir desse estado. Se o resultado da procura for vazio, então a procura não encontrou solução e procede-se á **geração** do próximo estado.

A lista de acções final é construída no retrocesso recursivo, através da composição das acções que geraram os vários estados no caminho do estado inicial até ao estado solução. Por último, é de notar que a fronteira na realidade não existe – esta é completamente implícita pela recursão.

2.2 Procura-A*

Tal como a função *procura-pp*, a função *procura-A** também recebe um *problema* e devolve o caminho desde o estado inicial até ao estado solução.

Esta é uma implementação do algoritmo A* que por si é uma variante do algoritmo *best-first search* em que os nós são avaliados de acordo com uma função de avaliação $f(n) = g(n) + h(n)$, onde $g(n)$ é o custo de caminho desde o estado inicial até ao nó n e $h(n)$ é o custo de caminho desde n até ao estado solução mais próximo.

Para esta procura decidimos definir uma estrutura chamada *no*, que representa um nó na árvore de procura. Este é composto por um estado, o valor de f nesse estado e o caminho desde o estado **actual** até ao estado **inicial**. Instâncias desta estrutura são comparadas pelo seu valor de f através da função *no<*, que diz se um nó é menor que outro.

No início da procura, a fronteira é inicializada como uma lista cujo único elemento é o nó inicial. Esta é mantida por ordem decrescente do valor de f dos seus nós ao longo do algoritmo, pois estes são sempre inseridos ordenadamente. Assim, o nó com menor valor de f está sempre á cabeça da lista.

O bloco principal do algoritmo é um *loop* infinito que, a cada iteração:

1. obtém o primeiro nó da fronteira e, se esta for vazia, o algoritmo termina e devolve *nil*;
2. verifica se o nó obtido contém um estado solução e, se sim, o algoritmo termina e devolve o caminho desde o estado **inicial** até ao estado **solução**;
3. expande o nó e adiciona os nós resultantes à fronteira.

¹ Isto significa que um estado não é gerado até que os estados já gerados na mesma expansão sejam processados.

O algoritmo não mantém conjunto de estados explorados nem outro tipo de optimizações porque para esta questão concentrámo-nos apenas em criar uma implementação concisa e simples de compreender.

3 Funções Heurísticas e de Custo de Caminho

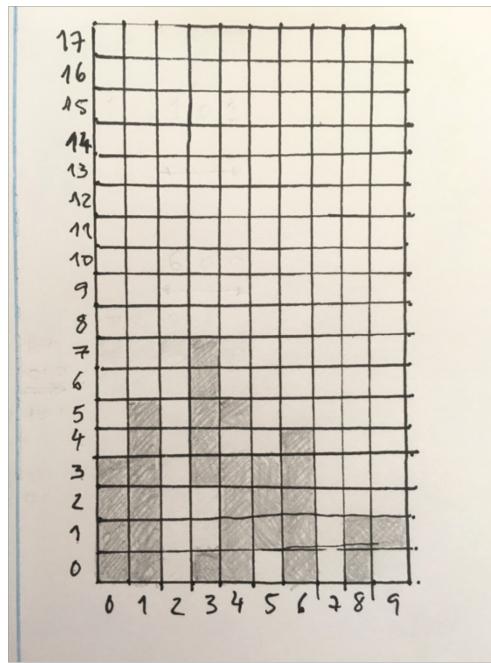


Figura 3-1 – Tabuleiro utilizado para exemplificar o cálculo de algumas das funções de custo de caminho/heurísticas.

3.1 Função *heuristica-pontuacao-1*

3.1.1 Motivação

A motivação para esta heurística segue da ideia de que, se um estado solução for aquele com maior pontuação possível², estamos mais perto do estado solução quanto maior for a nossa pontuação. Esta é calculada pela diferença entre um limite superior para a pontuação máxima e a pontuação obtida até então.

Para calcular o valor desta heurística num estado são necessários todos os seus atributos.

Esta heurística teve duas variantes ao longo do desenvolvimento do projecto: uma inicial (*heuristica-pontuacao-2*) e uma final (*heuristica-pontuacao-1*). A heurística final difere da inicial por ser calculada através de um limite superior mais apertado para a pontuação máxima, mas para tabuleiros inicialmente vazios, ou seja, para um sub-problema daquele que queremos realmente resolver.

A partir de agora, quando nos referirmos à *heuristica-pontuacao*, estaremos a referir-nos à *heuristica-pontuacao-1*.

3.1.2 Forma de Cálculo

Dado um conjunto de peças a ser colocado e um tabuleiro inicialmente vazio. Ora, a pontuação máxima que se pode obter (numa situação ideal) é através da eliminação do número máximo de blocos de quatro linhas possível. Como cada peça ocupa quatro posições no tabuleiro (daí as peças se chamarem *tetrominos* e o jogo *Tetris*) e o tabuleiro tem uma largura de dez posições, precisamos de (pelo menos)

² Na realidade, um estado solução é todo aquele que, não tendo o topo preenchido, não tem mais peças por colocar.

dez peças para eliminar um bloco de quatro linhas, ou seja, dez peças para obter 800 pontos, ou seja, 1 peça para cada 80 pontos.

Assim, dado um estado e sendo n a soma do número de peças total, ou seja, o número de peças por colocar mais o número de peças já colocadas desse estado, o valor desta heurística é dado por $80n - p$, onde p é a pontuação já obtida.

3.2 Função heuristica-altura

3.2.1 Motivação

A motivação para esta heurística segue da ideia de que, para estados consecutivos na árvore de procura, uma diminuição da altura da coluna mais alta do tabuleiro implica eliminação de linhas entre os estados³. No entanto, isto não se aplica a estados não consecutivos e, por isso, acabámos por não utilizar esta heurística na versão final do *procura-best*.

Para calcular a altura da coluna máxima apenas precisamos do tabuleiro contido no estado.

Esta heurística está intimamente relacionada com a função *custo-altura-agregada*, que descrevemos mais adiante.

3.2.2 Forma de Cálculo

O cálculo do valor desta heurística é muito simples. Dado um estado, devolve-se a altura da coluna mais alta do seu tabuleiro. No caso da Figura 3-1, esta função devolveria o valor 8, que é a altura da coluna 3.

3.3 Função custo-altura-agregadaⁱ

3.3.1 Motivação

Esta função segue de uma evolução da motivação inicial para a função *heuristica-altura*. Intuitivamente, quanto menores as alturas de todas as colunas, mais “controlado” está o jogo. No entanto, o efeito da utilização desta heurística é maior quanto maior for a duração do jogo, ou seja, quantas mais forem as peças a colocar inicialmente. Como nos foi dito o *procura-best* com tabuleiros com quatro a seis peças por colocar, esta função acabou por não figurar na função de avaliação final dessa procura.

Para calcular o valor desta função dado um estado, apenas é necessário o tabuleiro.

3.3.2 Forma de Cálculo

Para computar a altura agregada de um tabuleiro simplesmente tomamos a soma das alturas de todas as suas colunas. No caso da Figura 3-1, temos $4 + 6 + 0 + 8 + 6 + 4 + 5 + 0 + 3 + 3 = 39$.

3.4 Função custo-buracos^j

3.4.1 Motivação

Dado um tabuleiro, um buraco é definido como sendo uma posição não preenchida no tabuleiro onde há, no mínimo, uma segunda posição preenchida na mesma coluna e por cima da primeira.

A motivação para esta função é a seguinte: uma linha com buracos é mais difícil de eliminar, pois para preencher os buracos é preciso primeiro eliminar linhas que preencham esses buracos.

³ Na realidade, qualquer coluna serviria, desde que sempre a mesma, pois as linhas são eliminadas como um todo. No entanto, isso introduziria um enviesamento contra essa coluna.

Mais uma vez, apenas é necessário o tabuleiro do estado para calcular o valor da função.

3.4.2 Forma de Cálculo

Mais uma vez, o processo de cálculo é simples: dado um estado, o valor da função é dado contando o número de buracos no seu tabuleiro. No caso da Figura 3-1, temos dois buracos na coluna 3, um na coluna 4, um na coluna 5 e outro na coluna 9, para um total de 5 buracos.

3.5 Função *custo-peças-colocadas*

3.5.1 Motivação

A motivação para esta função é muito simples de compreender: dado um estado, quanto mais peças tiverem sido colocadas, mais “longe” estamos do estado inicial.

Dado um estado, para calcular o valor desta função apenas precisamos da lista de peças já colocadas.

3.5.2 Forma de Cálculo

O processo de cálculo é trivial: dado um estado, o *output* é o comprimento da sua lista de peças já colocadas.

3.6 Função *custo-relevo*ⁱ

3.6.1 Motivação

Dado um estado, esta função dá-nos uma medida do “relevo” do seu tabuleiro, ou seja, uma medida da variação entre as alturas entre colunas consecutivas no tabuleiro. Para isso, a função apenas precisa do tabuleiro em si.

A motivação segue da ideia de que, intuitivamente, um tabuleiro com menor relevo está mais “controlado”, pois terá menos “poços”. Dizemos que uma certa coluna do tabuleiro é um poço quando a sua altura é muito menor do que a(s) da(s) sua(s) coluna(s) adjacentes. Quando um poço tem uma diferença de altura superior a três posições relativamente às suas colunas adjacentes, torna-se impossível preenche-lo com uma só peça.

Esta função é um complemento para a função custo-altura-agregada. Com a função custo-altura-agregada, é possível que uma procura informada tenda para estados com tabuleiros com mais poços, algo que é balanceado pela função custo-relevo. No entanto, pela mesma razão da função custo-altura-agregada, a função custo-relevo acabou por não ser incluída na função de avaliação utilizada no *procura-best*.

3.6.2 Forma de Cálculo

Esta é função com o método de cálculo mais complicado. Dado um estado, o valor da função é dado tomando a soma dos valores absolutos das diferenças das alturas entre todas as colunas adjacentes. No caso da Figura 3-1, da esquerda para a direita, temos $|4 - 6| + |6 - 0| + |0 - 8| + |8 - 6| + |6 - 4| + |4 - 5| + |5 - 0| + |0 - 3| + |3 - 3| = 24$.

4 Estudo Comparativo

4.1 Estudo Algoritmos de Procura

4.1.1 Critérios a analisar

Os algoritmos foram comparados pelos seguintes critérios:

- **Qualidade das jogadas** – qual dos algoritmos, sob as mesmas circunstâncias, consegue a média das pontuações mais elevada?
- **Tempo de execução** – qual dos algoritmos chega a um estado solução em menos tempo?

4.1.2 Testes Efectuados

Foram efectuados três vagas de 100 testes. Para cada teste foi testado um par diferente (tabuleiro, peças por colocar), mas os testes foram os mesmos para as três vagas.

As peças foram computadas utilizando a função *random-peças*, disponibilizada pelo corpo docente.

A computação dos tabuleiros requereu um pouco mais de engenho. Foi também disponibilizada pelo corpo docente a função *cria-tabuleiro-aleatorio*. No entanto, os tabuleiros gerados não correspondem a tabuleiros típicos de um jogo de *Tetris*, mesmo ajustando os parâmetros da função.

Para criarmos os tabuleiros utilizámos uma versão alterada do algoritmo *procura-best* (descrito em pormenor na secção 4.3) onde utilizámos como função de solução uma nova função *solução'* que, dado um estado, devolve verdade se a aplicação da função *solução* original a esse estado devolver verdade e **se a sua pontuação for nula**; caso contrário, devolve falso. Assim, evita-se que se eliminem linhas durante a criação dos tabuleiros. Para os tabuleiros gerados não terem colunas muito altas nem com muitos buracos, a função de avaliação utilizada foi a soma das funções *custo-altura-agregada* e *custo-buracos*. Os tabuleiros foram criados através da colocação de conjuntos aleatórios de oito peças.

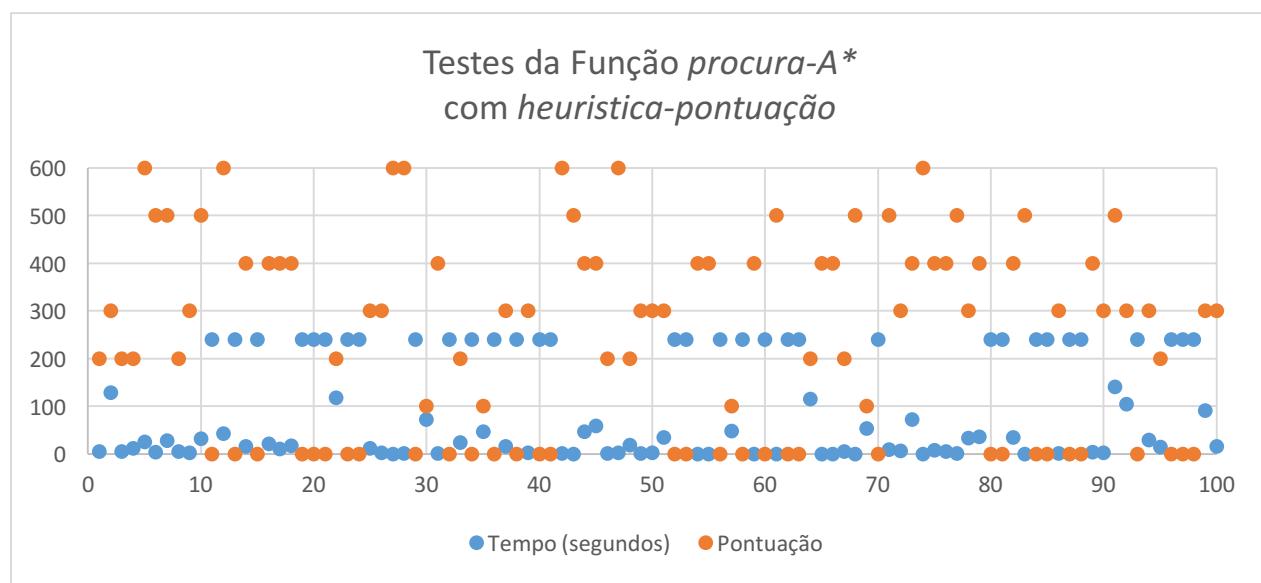
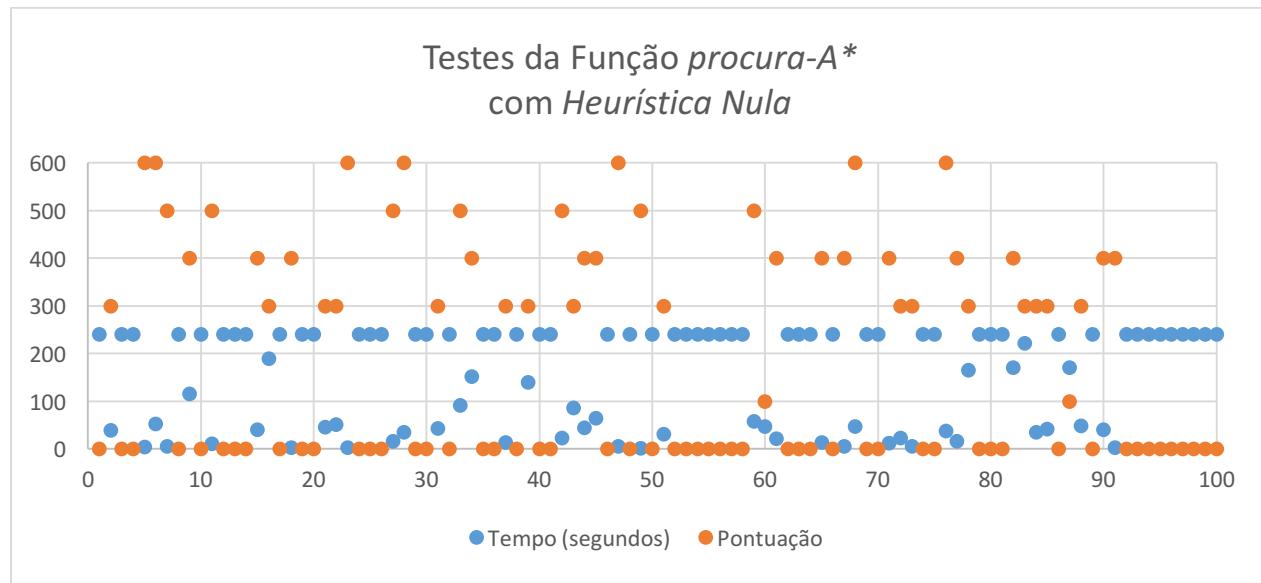
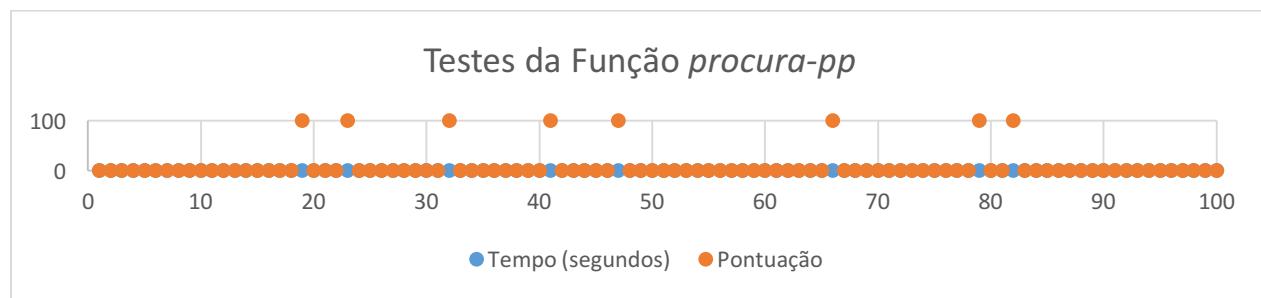
Na primeira vaga foi testada a implementação do procura-pp. Nas segunda e terceira vagas foram testadas duas procura A*: a primeira com heurística nula (devolve zero para qualquer input) e a segunda com a heurística-pontuacao-1.

Para limitar o tempo de execução dos testes das procura A*, o número de peças por colocar em cada teste foi sempre igual a quatro, nas três vagas. Para além disso, o tempo de execução das procura A* foi limitado a quatro minutos por teste (se a procura exceder o tempo limite, a pontuação obtida é zero).

Para as procura A*, a função de custo de caminho utilizada, *g*, foi sempre a função *custo-oportunidade*.

Os tempos foram calculados recorrendo à macro *get-time*.

4.1.3 Resultados Obtidos



Nota: Nos eixos dos xx estão representados os testes.

	<i>procura-pp</i>	<i>Procura-A* com heurística nula</i>	<i>Procura-A* com heuristica-pontuacao</i>
Média dos tempos (s)	0,00	154,55	95,76
Média das pontuações	8	183	241

Tabela 1 - Médias dos tempos e das pontuações obtidas nos testes dos algoritmos de procura.

4.1.4 Comparação dos Resultados Obtidos

Tal como seria de esperar, a *procura-pp* é simultaneamente a procura mais rápida e com menos qualidade. Isto acontece porque este algoritmo coloca as peças no tabuleiro na primeira configuração válida que encontrar, sem ter qualquer consideração pela pontuação obtida.

Por outro lado, as duas procuras A* diferem algo significativamente tanto nos tempos de execução como nas pontuações, com vantagem para a com *heuristica-pontuacao*. Isto seria de esperar, pois esta heurística vai fornecendo informação ao algoritmo A* sobre a distância até à solução, ao contrário da heurística nula.

4.2 Estudo funções de custo/heurísticasⁱ

4.2.1 Critérios a analisar

Este estudo foi feito tendo a função *procura-best* em mente. Com efeito, os testes foram efectuados com a própria função. Tal como discutido mais adiante, a *procura-best* demora sempre o mesmo tempo a executar e, por isso, já não faz sentido utilizar o tempo como critério de análise. Assim, o único critério utilizado neste estudo foi a **qualidade das jogadas**.

4.2.2 Testes Efectuados

Para testar as funções de custo/heurísticas e tomar uma decisão sobre elas para a *procura-best*, decidimos implementar uma função de avaliação, f , que resulta de uma combinação linear de todas as nossas funções de custo/heurísticas especificadas neste relatório e da função *custo-oportunidade* (especificada no enunciado). Dado um estado, e , temos

$$f(e) = \sum_{x \in F} c_x \cdot x(e)$$

onde F é o conjunto das funções de custo/heurísticas e c_x é um coeficiente associado à função x .

Para determinar as “melhores” constantes da combinação linear, decidimos recorrer a um algoritmo genético (descrito em pormenor na secção 4.4) O algoritmo foi corrido durante trinta e oito gerações, tendo sido inicializado com uma população de cem conjuntos de constantes e aplicado vinte testes por indivíduo por geração.

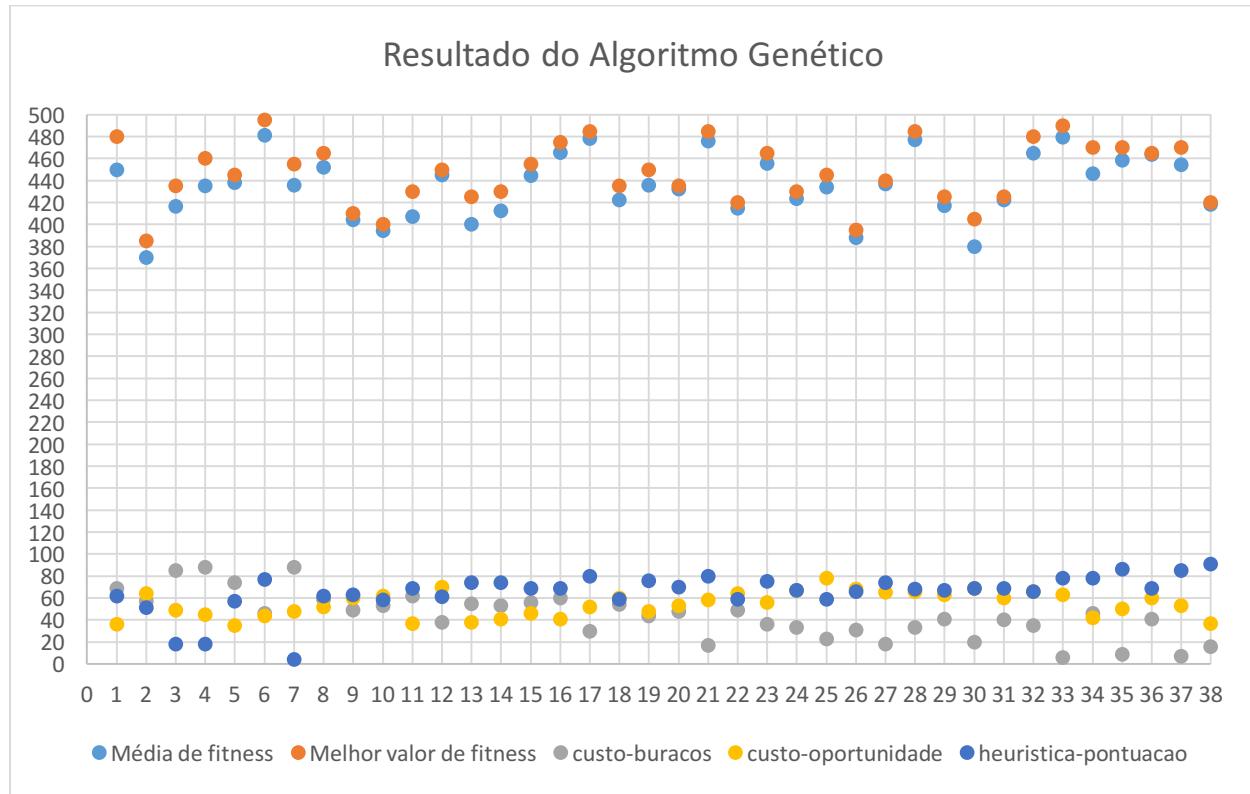
No entanto, o algoritmo genético demora mais tempo a convergir para uma solução quanto maior for a dimensão dos vectores da população. Posto isto, decidimos retirar do conjunto F as seguintes funções:

1. *custo-altura-agregada*;
2. *custo-pecas-por-colocar*;
3. *custo-relevo*;
4. *heuristica-altura*.

A justificação para o corte das funções 1, 3 e 4 já foi dada aquando da sua especificação. Para a eliminação da função 2, após testes à semelhança dos efectuados nas vagas 2 e 3 do estudo anterior, encontrámos justificação no facto de esta função não beneficiar as procura.

Restam-nos, então, as funções *custo-buracos*, *custo-oportunidade* e *heuristicapontuação*.

4.2.3 Resultados Obtidos



Notas:

- As gerações estão representadas no eixo dos xx.
- “Média de *fitness*” é a média dos valores de *fitness* em cada geração.
- “Melhor valor de *fitness*” é o valor do indivíduo com valor de *fitness* mais elevado (mais “fit”).
- “custo-buracos”, “custo-oportunidade” e “heuristicapontuação” correspondem às constantes da combinação linear associadas ás funções com o mesmo nome.
- As constantes foram todas multiplicadas por 100, para se notarem mais no gráfico. Isto não tem influência na qualidade da procura (como será explicado na secção 4.4).

4.2.4 Comparação dos Resultados Obtidos

Os valores de média e de melhor valor de *fitness* figuram nos resultados a título de curiosidade. Embora a qualidade dos indivíduos da população vá aumentando ao longo das gerações, estes valores dependem muito dos testes efectuados em cada geração. Pode se dar que uma população mais *fit* obtenha valores mais baixos que uma menos *fit*, porque os testes na sua geração eram mais “difíceis”.

O algoritmo genético faz a “comparação” entre as funções de custo/heurísticas por nós, atribuindo-lhes medidas de “importância” (constantes da combinação linear). Pelos resultados, conclui-se que as três

funções testadas têm diferentes importâncias, sendo a heurística-pontuação aquela com maior impacto. No entanto, não devemos descartar as outras heurísticas logo sem uma consideração cuidadosa das mesmas. Se elas não permitissem obter melhores resultados as suas constantes seriam (idealmente) zero. No final, nenhuma das constantes está próxima o suficiente de zero para as considerarmos inúteis.

Posto isto, escolhemos como nossa função de avaliação final (para ser usada no *procura-best*) a combinação linear das funções *custo-buracos*, *custo-oportunidade* e *heuristica-pontuacao* com constantes 16, 37 e 91, respectivamente.

4.3 Escolha da função *procura-best*

4.3.1 Motivação

A procura A* garante optimalidade se a heurística utilizada for admissível, no caso do TREE-SEARCH, ou consistente, no caso do GRAPH-SEARCH. É também conhecida por ser rápida, embora, por vezes, o seu consumo de memória possa ser proibitivo. Seria, á partida, uma boa aposta como algoritmo de procura a utilizar no *procura-best*.

No entanto, verificámos empiricamente que a nossa implementação do algoritmo A* não devolvia uma solução que cumprisse os requisitos de tempo exigidos⁴ para a maior parte dos problemas com quatro a seis peças por colocar. Assim, tornou-se necessário encontrar uma alternativa.

A procura A* não devolve solução em tempo útil porque não “decide” que caminho tomar na árvore de procura em tempo útil. Ora, isto deve-se ao enorme factor de ramificação da árvore⁵. Assim, podemos cortar no tempo de execução do algoritmo se o obrigarmos a tomar uma decisão mais cedo.

4.3.2 Implementação

Com efeito, o nosso algoritmo *procura-best* é uma versão modificada da nossa implementação *procura-A**. No *procura-best*, a cada x expansões de nós o algoritmo efectua uma selecção dos y melhores nós da fronteira. Dependendo dos valores de x e de y , o número de nós explorados pode reduzir-se substancialmente.

Esta solução não é óptima, pois é possível que se esteja a descartar um nó que nos levaria até à melhor solução. Aliás, o algoritmo nem sequer garante que se encontre uma solução!⁶ No entanto, verifica-se (empiricamente) que é praticamente certo que este algoritmo encontre solução, dados valores “adequados” de x e de y .

Na versão final do *procura-best*, são aplicadas várias iterações desta “procura A* modificada” dentro de um tempo limite de 19 segundos. No fim de cada iteração, mantém-se a melhor solução encontrada até então e duplicam-se os valores de x e de y . Após 19 segundos, o algoritmo termina e devolve a melhor solução encontrada.

⁴ Leia-se “em menos de 30 segundos”.

⁵ Este factor, b , varia com a peça a ser colocada, tendo um mínimo na peça O ($b = 9$) e um máximo na peça T ($b = 34$).

⁶ No entanto, é garantido que o algoritmo termina, pois o número de peças a colocar no tabuleiro é finito.

Como nota final, uma optimização **possível** seria manter um conjunto de nós explorados para eliminar expandir duas vezes um mesmo nó. Este seria implementado como uma *hash table* com elementos cujas chaves poderiam ser os seus valores da função de avaliação.

4.4 Implementação do Algoritmo Genéticoⁱ

Segue-se uma descrição pormenorizada do algoritmo genético implementado. O algoritmo é parametrizado pelo número de testes por indivíduo por geração, pelo tamanho da população, pelo tempo limite de cada teste e pelas heurísticas da combinação linear.

Os testes são os mesmos para todos os indivíduos de cada geração (mesmos pares (tabuleiro, peças por colocar)). Os tabuleiros utilizados são da *pool* de tabuleiros (pré-)computados por nós.

4.4.1 População

Os indivíduos da população são vectores em \mathbb{R}^3 da forma $\vec{v} = [a \ b \ c]^T$, sendo a , b e c as constantes relativas às funções *custo-buracos*, *custo-oportunidade* e *heuristica-pontuacao*, respectivamente.

Geralmente seria necessário pesquisar todo o espaço de soluções (\mathbb{R}^3) pelo vector de parâmetros “ideais”. No entanto, como a função de avaliação é uma combinação linear, o resultado de uma procura com essa função é invariante com a sua escala (ou seja, multiplicar todas as constantes da combinação pelo mesmo valor não faz com que a procura devolva uma solução diferente). Assim, apenas temos que considerar os pontos sobre uma esfera a três dimensões de raio unitário e centro na origem.

4.4.2 Função de *Fitness*

Dado um indivíduo a função de *fitness* corre os testes sobre o mesmo e devolve a média das pontuações obtidas.

4.4.3 Selecção⁷

O processo de selecção é implementado da seguinte forma:

1. A função de *fitness* é avaliada para cada indivíduo. Os valores obtidos são depois normalizados, ou seja, divide-se o valor de *fitness* de cada indivíduo pela soma dos valores de todos os indivíduos da população.
2. A população é ordenada por valores descendentes de *fitness*.
3. Procede-se à computação de valores acumulados de *fitness*. O valor acumulado de um indivíduo é a soma do seu valor de *fitness* normalizado com os valores dos indivíduos anteriores. Como os valores foram normalizados no ponto 1, o valor acumulado do último indivíduo é igual a 1.
4. Escolhe-se um valor aleatório, R , entre 0 e 1. O primeiro indivíduo com valor acumulado maior que R , é escolhido.

A este tipo de selecção dá-se o nome de *fitness proportionate selection*.

A cada duas selecções dá-se um processo de reprodução, produzindo dois novos indivíduos. São efectuadas tantas iterações deste processo quantas necessárias para criar uma nova população.

⁷ Este algoritmo foi baseado na página [https://en.wikipedia.org/wiki/Selection_\(genetic_algorithm\)](https://en.wikipedia.org/wiki/Selection_(genetic_algorithm)).

4.4.4 Reprodução

Dados dois vectores “pais”, x e y , são gerados dois vectores “filhos” através de uma operação de *one-point crossover*. Para isso, escolhe-se um ponto onde dividir os vectores pais, chamado o ponto de *crossover*, e combina-se a primeira metade do vector x com a segunda parte do vector y , e vice-versa.

Como resultado desta operação, os vectores pais serão substituídos pelos vectores filhos na próxima iteração do algoritmo.

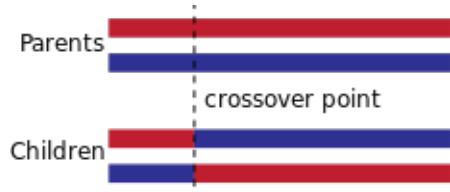


Figura 4-1 - Exemplificação da operação de one-point crossover⁸.

4.4.5 Mutação

Cada novo descendente tem uma pequena probabilidade (5%), de sofrer uma mutação. Uma mutação é caracterizada pela alteração aleatória de uma das componentes do indivíduo por um valor entre $\pm 0,2$.

No final de todo o processo, o vector é normalizado (havendo ou não mutação).ⁱ

ⁱ O nosso procedimento neste projecto foi altamente influenciado pelo artigo na página <https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/>. Em particular, as ideias para as funções *custo-altura-agregada*, *custo-buracos* e *custo-relevo*, assim como a ideia de aplicar um algoritmo genético para obter as constantes da combinação linear, foram retiradas desta página.

⁸ "OnePointCrossover" by R0oland - Own work. Licensed under CC BY-SA 3.0 via Commons - [https://commons.wikimedia.org/wiki/File:OnePointCrossover.svg](https://commons.wikimedia.org/wiki/File:OnePointCrossover.svg#/media/File:OnePointCrossover.svg)