



Inteligência Artificial

1.º Semestre 2015/2016

IA-Tetris

Relatório de Projecto

Realizado Por:

Luís Henriques, 77919

Luís Nunes, 77940

Ricardo Rei, 78047

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 solução	4
1.2.2. Função custo-caminho	4
1.2.3. Função accoes	4
1.2.3.1. Função gera-accoes	5
1.2.4. Função resultado	5
1.2.4.1. Função remove-linhas-e-atualiza-pontos	5
1.2.4.2. Função actualiza-tabuleiro	5
1.2.4.2.1. Função auxiliar coloca-peca-entre	5
1.2.4.2.2. Função auxiliar pivot	7
1.2.4.2.3. Função auxiliar conta-nils-da-coluna	7
2. Implementação Algoritmos de Procura	8
2.1. Procura-pp	8
2.2. Procura-A*	8
2.3. Algoritmos de Procura Local	9
2.3.1. Função escolhe-melhor-filho	9
2.3.2. Função escolhe-melhor-neto	9
2.3.3. Função escolhe-K-melhores	9
2.4. Algoritmo Genético	10
3. Funções Heurísticas	12
3.1. Heurística Somatório das Alturas	12
3.1.1. Motivação	12
3.1.2. Forma de cálculo	12
3.2. Heurística Número de Linhas Completas	12
3.2.1. Motivação	12
3.2.2. Forma de Cálculo	13
3.2.3. Heurística Linhas-completas-2níveis	13
3.2.3.1. Motivação	13
3.2.3.2. Forma de Cálculo	13
3.2.4. Formato Final da Heurística Linhas-Completas	13
3.3. Heurística Monotonia	14
3.3.1. Motivação	14
3.3.2. Forma de Cálculo	14
3.4. Heurística de Número de Buracos	15
3.4.1. Motivação	15
3.4.2. Forma de Cálculo	15
3.5. Meta-Heurística	15
3.5.1. Motivação	15
3.5.2. Forma de Cálculo	16

4. Estudo Comparativo.....	17
4.1. Estudo Algoritmos de Procura	17
4.1.1. Critérios a Analisar	17
4.1.2. Testes Efetuados	17
4.1.3. Resultados Obtidos	17
4.1.4. Comparação dos Resultados Obtidos	18
4.2. Estudo Funções do Custo/Heurísticas	19
4.2.1. Critérios a Analisar	19
4.2.2. Testes Efetuados	19
4.2.3. Resultados Obtidos	20
4.2.4. Comparação dos Resultados Obtidos	22
4.3. Escolha da Procura-best	22

1 Implementação Tipo Tabuleiro e Funções do problema de Procura

1.1 Tipo Abstracto de Informação Tabuleiro

Como representação do tipo abstracto tabuleiro o grupo decidiu a utilização de um simples array 18x10. No entanto este apresenta-se invertido face as linhas, ou seja, a linha 0 do array corresponde à linha 17 do tabuleiro e vice-versa. Face às colunas não há qualquer inversão.

A escolha desta implementação foi feita devido à sua facilidade de representação uma vez que ao aparecer no terminal o array já tinha o formato final de um tabuleiro e não era necessário pensarmos nele como estando invertido.

Em termos de manipulação o único cuidado que é necessário é relativo às funções: *tabuleiro-preenchido-p*, *tabuleiro-altura-coluna*, *tabuleiro-preenche!* e *tabuleiro-remove-linha!*. Em todas estas funções quando queremos aceder à *i*-ésima linha precisamos de considerar que em termos do array isso corresponde à linha $(17 - i)$.

Como alternativas à implementação escolhida poderíamos ter usado um array simples sem que as linhas tivessem invertidas ou então uma estrutura. Em relação à primeira achámos que a representação não é tão intuitiva e não compensava a facilidade de manipulação. Em relação à segunda achámos que estaríamos a complicar algo que se adequava perfeitamente a uma estrutura base da linguagem.

1.2 Implementação de funções do problema de procura

Para o problema de procura do IA-Tetris as funções mais relevantes são as funções: *accoes* e *resultado*, que iremos descrever em detalhe. As funções: *solução* e *custo-caminho*, não serão tão relevantes e como tal não lhes vamos dedicar a mesma atenção, dando apenas uma descrição breve sobre estas.

1.2.1 Função solucao

Esta função verifica se um estado é um estado final de acordo com o que é definido no enunciado. Tem uma implementação simples que recebe um estado e realiza uma condição *if* com uma operação lógica *and* que verifica duas condições: verifica se o estado é final (através da chamada da função *estado-final-p*) e se o topo do tabuleiro não está preenchido (através da chamada à função: *tabuleiro-topo-preenchido-p* a que vamos negar o resultado).

1.2.2 Função custo-caminho

Esta função apenas serve para saber o custo de ir de um estado para o outro através da realização de uma acção. Pode variar bastante de acordo com o objectivo que queremos atingir e portanto não iremos entrar em detalhes das, diversas, possíveis implementações, nesta secção. Como inicialização por omissão usámos a função: *custo-oportunidade*.

1.2.3 Função accoes

Esta função é uma das mais importantes uma vez que nos dá as possíveis transições entre estados quando estamos a fazer a procura.

Como argumentos vai receber um estado e caso ele não seja um estado final (condição verificada na primeira linha do corpo da função através de um *if* que chama a função *estado-final-p* para o estado recebido como argumento) vai gerar todas as acções possíveis para a próxima peça nas *pecas-por-colocar* do estado recebido.

Para gerar todas as acções para a próxima peça precisamos de primeiro identificar a mesma. A peça é obtida através de um *car* na lista de *pecas-por-colocar* e depois utilizado um case que vai identificar a peça. Depois chamamos a função auxiliar: *gera-accoes* para todas as rotações da mesma devolvendo uma lista que é a soma das listas retornadas pela função auxiliar.

1.2.3.1 Função gera-accoes

Esta função recebe uma configuração e devolve as acções possíveis para essa mesma configuração. Em termos de implementação isto é conseguido através de um loop que vai de 0 até à coluna ¹(11 - dimensão-lateral-da-configuração), criando diversas acções que começam na i-ésima coluna e contêm a configuração recebida. Depois de criada a nova acção vai guardá-la numa lista. No final retorna a lista com todas as acções.

1.2.4 Função resultado

Esta função em relação à primeira entrega realizada é a mais complexa, utilizando uma série de funções auxiliares. Em relação ao problema da procura do IA-Tetris tem uma importância muito grande uma vez que é com ela que chegamos aos novos estados através de uma dada acção.

A função recebe um estado e uma acção, começando por criar um novo estado (copia do primeiro) sobre o qual vai trabalhar, de forma a preservar o estado recebido. De seguida transfere a primeira peça das *pecas-por-colocar* para o início das *pecas-colocadas* e chama a função: *actualiza-tabuleiro* (secção 1.2.4.2) que vai desenhar no tabuleiro do novo estado a acção recebida. Por último, se o novo estado não tiver o topo preenchido vamos chamar a função: *remove-linhas-e-actualiza-pontos* (secção 1.2.4.1). No final devolve-se o novo estado após as alterações que lhe foram sendo feitas.

1.2.4.1 Função remove-linhas-e-actualiza-pontos

A função *remove-linhas-e-actualiza-pontos* recebe um estado que resulta de aplicar uma acção sem qualquer tratamento posterior em termos de pontos ou linhas completas. Como tal vai iniciar uma variável *numero-linhas-removidas* e realizar um loop de 0 a 17. Dentro desse loop vamos usar uma condição *if* que chama a função *tabuleiro-linha-completa-p* para verificar se a i-ésima linha está completa. Caso esteja completa vamos remover essa linha através da função *tabuleiro-remove-linha!* e incrementar a variável *numero-linhas-removidas*. Por fim é necessário decrementar a variável responsável pela iteração de forma a mantermo-nos na mesma linha, visto que ao remover a linha *i* todas as linhas descem.

1.2.4.2 Função actualiza-tabuleiro

Esta função vai receber como argumentos um tabuleiro e uma acção e vai devolver o resultado da aplicação dessa acção no tabuleiro. Para melhor explicar esta função vou separar o raciocínio em diferentes partes e introduzindo as diferentes funções auxiliares.

1ª- Parte:


Motivação - como colocar uma peça entre uma dada linha e coluna? Ou seja partindo por exemplo da coluna 2 e linha 2, como colocar a peça X nessa posição?

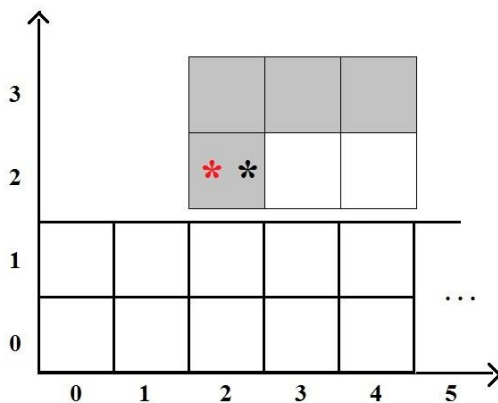
1.2.4.2.1 Função auxiliar coloca-peca-entre

Esta função recebe um tabuleiro, uma peça, um inteiro referente à altura/linha e um inteiro referente à coluna. De seguida inicia dois loops, um loop exterior que percorre a dimensão lateral da peça e um loop interior que para cada índice da dimensão lateral da peça vai percorrer a altura da mesma. No loop interior vamos ter uma condição *if* que apenas verifica se a peça naquela posição está preenchida, note-se que a representação de uma peça é sempre um array NxN e o que dá formato à

peça são as posições preenchidas dentro desse array, portanto se uma posição estiver preenchida só vamos ter de preencher o correspondente no tabuleiro.

Uma vez que recebemos a altura e a coluna como argumentos basta então preencher o tabuleiro na linha (altura + j) e coluna (coluna + i), onde j e i são o j-ésimo e i-ésimo índices correspondentes à linha e coluna da peça que estão preenchidos.

coloca-peca-entre (tabuleiro, , 2, 2)



* Na peça = posição[1][1]

* No tabuleiro = posição [2+1][2+1]

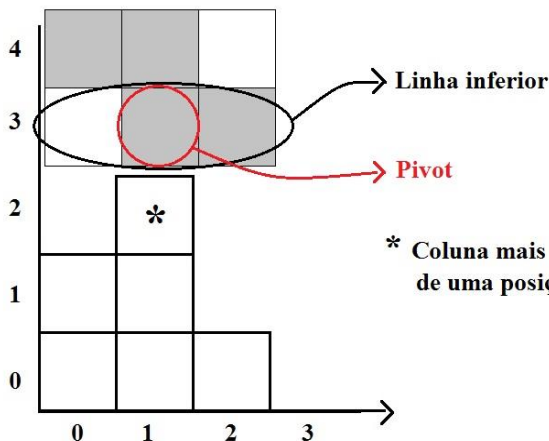
Imagem 1.1 - Coloca-peca-entre

2ª Parte:

Motivação - Tendo em conta a parte inferior da peça, ou seja a linha inferior, qual é o seu pivot?

Como pivot o grupo entendeu que será o índice da peça da linha inferior que assenta na coluna mais alta do tabuleiro no seu caso mais simples. Para ajudar a perceber esta definição é recomendado consultar a imagem a baixo.

pivot (tabuleiro, (0 . ))



* Coluna mais alta por baixo de uma posição da base

Imagem 1.2 – Pivot

1 Note-se que o tabuleiro tem as colunas contadas de 0 a 9 e a dimensão-lateral-da-configuração quando calculada através de: (second (array-dimensions configuracao)) da um valor a começar em 1 quando na verdade devia começar em zero para os índices baterem certo, daí a utilização do 11 que vai compensar essa diferença.

1.2.4.2.2 Função auxiliar pivot

Esta função recebe um tabuleiro e uma acção. Na peça respectiva aquela acção vai varrer a linha inferior, procurando as posições inferiores que estão preenchidas e escolhe aquela que tiver para aquele tabuleiro a coluna mais alta por baixo.

Em relação á implementação, utiliza uma variável índice que começa a NIL, começando por fazer um loop na linha inferior que vai ter duas condições dentro. A primeira que serve para inicializar a variável índice: *if posicao-ocupada* e índice = NIL então índice = i. E uma segunda para factor de desempate que verifica: *if posicao-ocupada* e [tabuleiro-altura-coluna (tabuleiro, *accao-coluna* (accao) + j) > tabuleiro-altura-coluna (tabuleiro, *accao-coluna* (accao) + índice)] então índice = i. No final retorna o índice.

3ª Parte:

Motivação - Quantas posições desocupadas tem uma peça na coluna que vai ficar por cima da coluna mais alta das colunas do tabuleiro entre a coluna dada pela acção e a coluna (dada pela acção + *dimensao-lateral-da-peca*)?

1.2.4.2.3 Função auxiliar conta-nils-da-coluna

Esta função recebe uma acção e uma coluna e vai fazer um loop que para na primeira posição ocupada dentro da configuração da peça para aquela mesma coluna. No final retornamos o número de iterações (dado pela variável que itera). Repare-se que as peças, tal como o tabuleiro, têm uma representação inversa em termos de linhas pelo que começar a iteração por zero é a mesma coisa que começar a iteração na parte de baixo da peça.

4ª Parte:

Por fim voltando a função *actualiza-tabuleiro*, começamos por guardar o índice da coluna mais alta do tabuleiro que se encontra entre a coluna dada pela acção e a coluna dada pela acção + *dimensao-lateral-da-peca*. Depois de sabermos qual é essa coluna vamos querer saber o pivot da peça e caso o pivot fique por cima da coluna mais alta, vamos apenas desenhar a peça partindo da coluna dada pela acção e a altura mais alta. Caso contrário iremos ter um encaixe.

Nos encaixes vamos querer saber o decremento na altura a que vamos desenhar a peça tendo em conta que inicialmente a altura a que iríamos desenhar a peça seria a altura da coluna mais alta. Começamos então por ver o numero de nils que existe na coluna que vai ficar por cima da coluna mais alta, depois decidimos o nosso decremento através do mínimo entre a coluna mais alta menos a coluna que vai ficar por baixo do pivot e o numero de nils. Por fim chamamos a função *coloca-peca-entre* com os seguintes argumentos: tabuleiro, peça, (altura mais alta - decremento), coluna dada pela acção.

2 Implementação Algoritmos de Procura

2.1 Procura-pp

A procura em profundidade primeiro é um dos mais conhecidos algoritmos para travessia ou de procura em árvores e grafos. Formalmente, um algoritmo de procura em profundidade realiza uma procura não informada que inicia a sua expansão na raiz da árvore de procura e vai progredir através da escolha do primeiro filho desse nó, repetindo este processo até se chegar a um estado objectivo ou até chegar a um nó sem filhos. Caso tenha chegado a um nó sem filhos o algoritmo retrocede (backtrack) e começa no próximo nó. Numa implementação não recursiva os nós são adicionados a uma pilha para realizar a exploração.

No âmbito deste projecto o algoritmo pp segue um critério “Last in First Out” respeitando a definição formal que utiliza uma pilha para manter a fronteira/abertos de nós (conjunto de nós por explorar).

Em relação à implementação do mesmo foi seguida e adaptada a implementação fornecida pelo repositório de código do livro “Artificial Intelligence: A Modern Approach: Stuart Russel and Peter Norvig 2003 Prentice Hall”.

2.2 Procura-A*

O algoritmo A* é um algoritmo bastante usado para a procura de caminhos. Devido à sua performance e comportamento assertivo tornou-se bastante popular e conhecido.

A* na sua génese usa um best-first-search e encontra o caminho de menor custo que vai do nó inicial até ao nó objectivo construindo uma árvore de caminhos parciais. Os nós a expandir na procura A* são guardados por uma ordem de prioridade dada por uma função de custo que combina o valor estimado por uma heurística com a distância ao nó inicial.

Função de custo: $f(n) = g(n) + h(n)$, onde $g(n)$ é o custo do nó inicial até ao nó n e $h(n)$ é a estimativa do custo de n até ao objectivo.

Face à implementação utilizada neste projecto é utilizada uma min-heap para manter a ordem de prioridade da fronteira/abertos. Desta forma o nó com valor $f(n)$ mais baixo é mantido sempre na raiz da heap bastando apenas retirar o primeiro elemento da heap para obtermos o próximo nó a ser explorado, e chamando a função *heapify* vamos mantendo as propriedades de min-heap.

Dado o critério de desempate LIFO caso os nós sejam avaliados com o mesmo valor $f(n)$ e visto que as min-heaps ao fazerem a inserção de nós vão manter sempre mais a cima o primeiro nó com o valor mínimo, o grupo decidiu que os sucessores de um nó expandido antes de serem colocado na heap eram invertidos, de forma a que o último sucessor com um valor X considerado mínimo seja mantido a cima de qualquer sucessor com o mesmo valor que tenha surgido antes.

Vantagens de utilização de uma min-heap:

Uma alternativa à utilização da heap era representar a fronteira/abertos com uma lista ordenada. Esta implementação implicava que sempre que fossem inseridos novos elementos toda a lista teria que ser ordenada de novo. Uma ordenação típica através de algoritmos como quicksort, etc., têm na melhor das hipóteses uma complexidade $O(n)$ (ex: insertion sort) sendo que os casos médios tipicamente rondam valores de $O(n \log n)$ (ex: quicksort).

A utilização de uma heap permite que a inserção tenha no pior caso uma complexidade $O(\log n)$, outras operações como extração também são de complexidade $O(\log n)$ no pior caso, o que é um pouco pior que manter uma lista ordenada onde bastava retirar o primeiro elemento mantendo-se a lista ordenada na mesma. Em suma usando uma heap temos complexidades sempre de $O(\log n)$ no pior caso enquanto que utilizando outras alternativas estamos sujeitos a complexidade maiores para casos específicos.

De forma geral a implementação usada foi a do repositório do livro como referido na procura-pp, à excepção de pequenas alterações.

2.3 Algoritmos de procura local

Foram implementados 3 algoritmos de procura local diferentes e todos eles utilizam critérios de escolha diferentes que levam ao próximo estado. Estes critérios são realizados pelas funções que vamos descrever de seguida. No entanto a procura local vai comportar-se de forma semelhante, chamando em loop enquanto existirem peças essas funções, passando-lhes um estado e uma função heurística. Por fim, através do valor retornado pelas mesmas são guardadas as decisões feitas e o estado retornado que vai ser utilizado para a próxima iteração do loop.

2.3.1 Função escolhe-melhor-filho

A função *escolhe-melhor-filho* torna o algoritmo de procura local um hill-climbing específico para o problema IA-Tetris. Esta função recebe um estado e gera os seus sucessores, de seguida escolhe um estado sucessor tal que o seu valor heurístico seja o mais baixo possível, note-se que caso haja vários estados com o mesmo valor mínimo a escolha é feita de forma aleatória.

Uma pequena optimização realizada foi que à medida que se efectua a expansão vamos guardar um nó com o valor heurístico mais baixo, poupando desta forma computações adjacentes que vão procurar esse mesmo mínimo. Desta forma no final da expansão basta retirar da lista de sucessores todos os nós com valores heurísticos maiores do que o valor do nó guardado e escolher aleatoriamente entre os que sobram.

2.3.2 Função escolhe-melhor-neto

A função *escolhe-melhor-neto* torna, à semelhança da função *escolhe-melhor-filho*, a procura local num hill-climbing a 2 níveis específico para o IA-Tetris. No tetris a utilização de um hill-climbing (típico) torna o cálculo de acções muito mais eficiente, mas talvez seja um bocado extremo visto que a expansão de um segundo nível é perfeitamente computável com um total de 900 nós novos (se considerarmos um branching aproximado de 30). Já o terceiro nível tem 27000 nós novos sendo portanto extremamente pesado expandir mais que 2 níveis. Tendo em conta estas considerações o algoritmo recebe um nó e expande gerando os sucessores, depois para cada sucessor faz o mesmo gerando os sucessores dos sucessores ou seja os nós netos. Depois avalia todos os netos e escolhe o neto com melhor valor heurístico ou então aleatoriamente entre os vários com o valor heurístico mínimo.

À semelhança da optimização realizada no algoritmo Escolhe-melhor-filho o Escolhe-melhor-neto também vai manter um candidato a melhor neto para que no final da expansão dos nós netos já saibamos um nó com valor heurístico mínimo.

2.3.3 Função escolhe-K-melhores

A função *escolhe-k-melhores* foi desenvolvida pelo grupo como tentativa de atingir melhores pontuações na procura local. Na procura local com a função *escolhe-melhor-neto* temos resultados bastante satisfatórios mas uma vez que só faz escolhas a 2 níveis, a probabilidade de fazer 3 ou 4 linhas de uma vez é pouca. Para a eliminação de 3 ou 4 linhas na maioria das vezes é necessário fazer um combinação específica de 3 ou mais peças. Desta forma a intuição desta função é escolher por nível apenas os K melhores nós para colocar na fronteira/abertos, esquecendo a existência de todos os restantes. Esta redução do branching permite fazer considerações até ao quarto nível de profundidade (4 peças). O k escolhido foi 5 visto que com k igual a 5 no nível 4 iremos ter cerca de 625 nós, que é um valor de ordem de grandeza semelhante a expandir toda a árvore até ao segundo nível (900 nós).

A implementação é simples, a função recebe um estado, cria um nó raiz e cria todos os sucessores aos quais faz sort de acordo com uma função heurística, tal que o nó com menor valor fique no final da lista. Depois retira da lista de sucessores todos os nós até sobrares os últimos 5, e para cada um desses 5 vai fazer o mesmo, criando a lista de sucessores do nível 2 que é a soma dos melhores 5 nós partindo de cada um dos 5 sucessores do nível 1. Vamos repetir este raciocínio até ao nível 4.

Apesar de inicialmente esta ideia parecer promissora acabámos por não a explorar muito visto que os primeiros testes realizados mostraram pontuações iguais á escolha local a 2 níveis, e portanto focamos o nosso esforço em melhorar as escolhas feitas a 2 níveis.

2.4 Algoritmo Genético

Este algoritmo não é directamente relacionado com a procura mas foi utilizado para melhorar as constantes associadas á meta-heurística utilizada para fazer a *procura-best* que é uma procura local a dois níveis de profundidade.

As constantes associadas às “features” tipicamente não são fáceis de adivinhar e como tal tentámos que o algoritmo genético as “aprendesse”.

População:

A população inicial é representada por um array de tamanho 200 em que cada entrada contém um vector com 4 constantes que vão de 0.0 a 1.0. Esses vectores representam os indivíduos da nossa população. Para cada indivíduo corremos 10 jogos de 16 peças aleatórias atribuindo no final dos 10 jogos um fitness. A escolha do número de peças foi 16 pois já permite num tabuleiro inicialmente vazio que se façam algumas linhas e ao mesmo tempo as acções demoram apenas 1.7 segundos a calcular. A escolha de 10 jogos é executada de modo a diminuir certas ambiguidades, como por exemplo testar um indivíduo X com várias peças “fáceis” (ex: uma serie de I’s) e um indivíduo Y com peças “difíceis” (ex: uma serie de Z’s e S’s).

Fitness dos individuo:

O fitness de cada indivíduo é igual ao somatório de pontos realizados nos 10 jogos. Quanto maior for este valor mais correto é o “trade-off” feito por um dado indivíduo em relação às diferentes heurísticas reflectindo-se em escolhas que levam a um maior numero de pontos.

Seleção para cruzamento:

Para cruzamento escolhemos uma amostra aleatória de 20 indivíduos da população inicial. Com base no fitness de cada indivíduo fazemos sort da amostra e escolhemos os 2 melhores para cruzar. Vamos repetir este processo até termos 60 novos indivíduos que vamos inserir na população inicial substituindo os 60 com pior fitness.

Função de Cruzamento:

A função de cruzamento recebe 2 indivíduos, um pai e uma mãe. Para cada entrada vai fazer uma “média” que tende para o progenitor com melhor fitness.

Formula para cada entrada:

Sendo X e Y os respectivos progenitores e W o resultado do cruzamento, cada entrada de W será calculada da seguinte forma:

$$W[i] = X[i] * (\text{fitness de X}) / (\text{fitness de X} + \text{fitness de Y}) + Y[i] * (\text{fitness de Y}) / (\text{fitness de X} + \text{fitness de Y})$$

Caso os fitness de X e Y sejam iguais basta fazer uma média.

Função Mutação:

Por cada indivíduo novo chamamos a função mutação. Esta função gera um valor aleatório entre 0 e 100 e caso esse valor seja menor que 6 é gerado outro valor aleatório entre 0 e 2, para decidir se se trata de uma mutação negativa ou positiva.

Uma mutação negativa é criada através da subtração de 0.2 em todas as entradas, uma mutação positiva é realizada somando o mesmo valor.

Note-se que uma mutação nunca pode tomar valores superiores a 1 ou inferiores a 0, portanto para esses casos o valor da entrada fica igual a 0.999999 ou 0.00001 respectivamente.

Esta função no entanto tem um pequeno erro visto que ao variar as entradas todas na mesma proporção cada heurística continua a ter a mesma influência em relação às outras, no entanto não chegámos a corrigir este erro, visto que só nos apercebemos dele um pouco tarde e as constantes calculadas já eram bastante satisfatórias.

Resumindo o algoritmo, este inicializa uma população e depois faz um loop de 0 a 19, que equivale a 20 gerações onde vai correr os tais 10 jogos de 16 peças. Depois com os fitnesses já atribuídos faz a selecção de amostras de tamanho 20 e cruza os 2 melhores até termos 60 novos indivíduos, que podem ou não sofrer mutações. A nova população é obtida através da inserção desses indivíduos na população substituindo os 60 piores. Por fim voltamos a fazer “set” dos fitness todos a 0 e repete-se o processo.

À medida que as gerações são geradas e antes de se dar “set” aos fitness é feito um “print” para o terminal com a média da população e o indivíduo que teve um fitness mais alto.

3 Funções Heurísticas

3.1 Heurística Somatório das Alturas

3.1.1 Motivação

A ideia associada a esta heurística é a seguinte: Na colocação das peças vamos tentar que o tabuleiro mantenha as alturas mais baixas possíveis para cada coluna, ou seja quanto mais perto tivermos da linha 0 mais longe estamos de perder e vice-versa.

Esta heurística ajuda-nos a perceber a qualidade do nosso estado, uma vez que à medida que vamos jogando a tendência é que a altura do agregado de peças também vá subindo. A única forma de contrariar este efeito é ir eliminando linhas e colocando as peças encaixadas umas nas outras.

Ao testarmos o algoritmo apenas com esta heurística reparamos que o seu comportamento singular se revelou relativamente satisfatório, visto que de forma a evitar o crescimento da altura iam-se eliminando algumas linhas. Pode ser visto na imagem a baixo o resultado de um jogo de 16 peças num tabuleiro vazio com esta heurística.

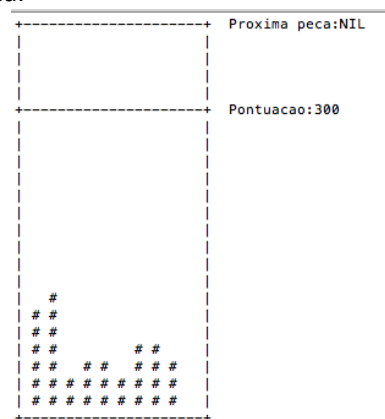


Imagem 3.1 – Resultado Heurística Somatório de Alturas para 16 peças

3.1.2 Forma de Cálculo

Dado um nó X aplicamos:

$$\sum_{k=0}^9 \text{altura da coluna } k, \text{ do tabuleiro do estado guardado por X.}$$

No entanto apesar de esta heurística conseguir resolver jogos com bastantes peças sem perder falta-lhe certas considerações que levem a um total de pontos mais elevado.

3.2 Heurística Numero de Linhas Completas

3.2.1 Motivação

Esta heurística é provavelmente a mais intuitiva para qualquer pessoa que já tenha experimentado o jogo. Como forma de avaliar a qualidade de um estado podemos ver quantas linhas foram completas na passagem do estado anterior a este. No entanto os resultados da utilização desta heurística singularmente não são satisfatórios uma vez que na ausência de oportunidade para completar uma linha a colocação de peças é feita de forma aleatória pelo algoritmo e como resultado da escolha aleatória também a probabilidade de aparecer linhas para completar é menor.

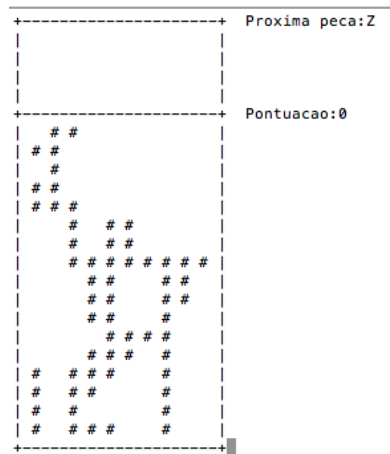


Imagem 3.2 – Resultado Heurística Numero de Linhas Completas para 16 peças

3.2.2 Forma de cálculo (original)

Dado um nó X:

Diferença = pontos do estado guardado em X - pontos do estado guardado pelo pai de X

se diferença = 100 => 1 linha

se diferença = 300 => 2 linhas

se diferença = 500 => 3 linhas

caso contrario 4 linhas

Retorna linhas*-1

A multiplicação por -1 é devido ao nosso algoritmo procurar os nós com os valores mais baixos dados pelas heurísticas e neste caso estamos a querer maximizar o número de linhas.

Em suma, esta heurística só por si diz muito pouco ou nada na maior parte dos casos, no entanto combinada com as outras irá ser um importante factor de desempate que nos vai levar à maximização dos pontos e nesse sentido o grupo decidiu fazer algumas alterações á ideia inicial que são explicadas de seguida.

3.2.3 Heurística linhas-completas-2niveis:

3.2.3.1 Motivação

Repare-se que o nosso algoritmo faz a escolha local a 2 níveis de profundidade. Isto implica que vamos querer avaliar a qualidade de passagem do estado inicial para os seus netos, como tal, se avaliarmos apenas as linhas completas por um neto em relação ao seu pai podemos estar a esquecer-nos de ter em consideração as linhas completas na passagem do estado inicial para esse mesmo pai. Devido à existência deste detalhe a heurística inicial é alterada para: Dado um estado X as suas linhas completas são a soma das linhas completas do pai e das suas.

3.2.3.2 Forma de cálculo

Dado um nó X

retorna (linhas-completas pelo pai de X) + (linhas-completas por X)

Ainda assim existe algo que esta heurística não considera.

3.2.4 Formato final da heurística linhas-completas

No caso de termos um estado X que a chamada da função linhas completas retorna 1 e para o seu pai também é retornado 1, então o resultado final será 2 o que resulta em 200 pontos.

Considere agora um estado Y no qual a chamada da função linhas completas para o pai retorna 0 e a chamada da função linhas-completas para Y retorna 2, sendo esse o resultado, no entanto os pontos serão 300. Com base nestes dois exemplos podemos concluir que a importância relativa ao número de linhas completas numa jogada não é considerada e como resolução o grupo decidiu modificar a heurística inicial para retornar o quadrado das linhas (multiplicado por -1 na mesma). Dessa forma voltando ao exemplo anterior Y retornaria 4 (2^2) e X retornaria 2 ($1^2 + 1^2$), dando prioridade ao estado que completa as linhas numa só jogada em vez dos que completam o mesmo número de linhas em duas jogadas.

3.3 Heurística Monotonia

3.3.1 Motivação

A existência desta heurística está relacionada com o conceito a que decidimos chamar poço. Um poço é uma coluna com uma altura muito inferior à das suas adjacentes, por exemplo, um tabuleiro que a coluna 3 tem altura 2 e a coluna 2 e 4 têm alturas de 10 é um tabuleiro mau pois à excepção da peça I quase nenhuma peça consegue preencher aquela coluna, dificultando o acto de completar linhas.

Assim sendo esta heurística dá-nos a variação entre as alturas de colunas adjacentes. Uma jogada que coloque as alturas todas iguais implica a realização de linhas.

Os testes a esta heurística de forma singular não foram no entanto muito satisfatórios uma vez que salvo raras excepções ela não completava linhas limitando-se apenas a encaixar peças para que a altura das colunas se mantivesse estável, como mostra a imagem, acabando por perder ao fim de algumas peças.

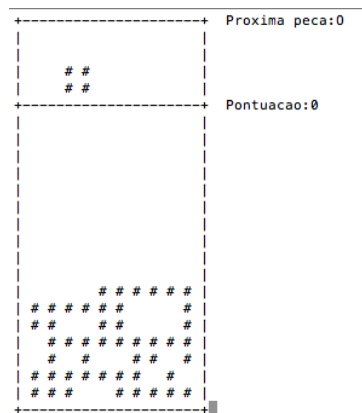


Imagem 3.3 – Resultado Heurística Monotonia para 16 peças

No entanto o comportamento de monotonia não deixa de ser um factor importante que deve ser tido em consideração quando se está a jogar Tetris, pois um tabuleiro monótono pode implicar que com 1 peça se eliminem uma série de linhas.

3.3.2 Forma de Calculo:

Dado um nó X:

$\sum_{k=0}^8 |altura\ coluna\ k - altura\ coluna\ k + 1|$, sendo a coluna k e k+1 colunas do tabuleiro do estado guardado no nó X

3.4 Heurística Numero de buracos

3.4.1 Motivação

Um buraco é uma posição desocupada numa coluna que tenha posições ocupadas por cima, por exemplo, imagine que a posição na linha 1 coluna 3 não esta ocupada e no entanto temos para a mesma coluna posições ocupadas em linhas superiores, então a posição (1,3) contem um buraco. A existência de buracos torna mais complicado limpar a linha onde este se encontra uma vez que vamos precisar de limpar as linhas superiores, ocupar esse mesmo buraco e só depois conseguimos eliminar essa linha.

Os testes a esta heurística de forma singular, à semelhança das outras heurísticas também não apresentam os melhores resultados. Esta heurística faz com que as peças encaixem na perfeição mas não tem qualquer preocupação com as linhas completas deixando o tabuleiro atingir alturas muito altas sem realizar qualquer pontuação, como podemos ver pela imagem em baixo que resulta de correr um jogo de 16 peças com base apenas nesta heurística.

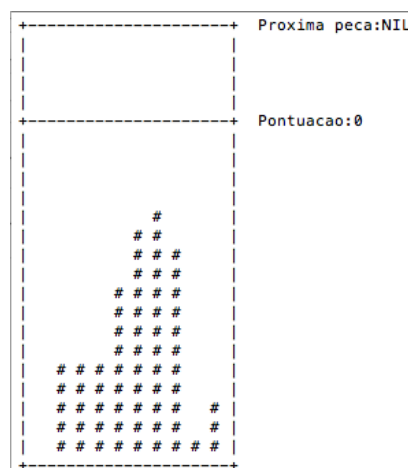


Imagem 3.4 – Resultado Heurística Numero de Buracos para 16 peças

3.4.2 Forma de cálculo

Dado um nó X:

buracos = 0

loop i de 0 a 9

altura = altura da coluna i

loop j de 0 a altura

se o tabuleiro não tiver preenchido na posição j,i e tiver preenchido na posição j,i+1
então, buracos = buracos + 1

3.5 Meta-Heurística

3.5.1 Motivação

Após conhecidas as características mais importantes de um tabuleiro no jogo de Tetris como é que as podemos combinar para que o nosso algoritmo consiga fazer melhor as suas escolhas, tendo em conta o “trade-off” entre elas?

Um factor importante a ter em conta é que estas heurísticas não são independentes. Um acontecimento comum é que sejamos obrigados a fazer um buraco para conseguir completar uma linha, ou que seja necessário criar um poço onde depois entra uma peça I de forma a eliminar 4 linhas.

Como tal decidimos combinar as diferentes funcionalidades dadas pelas heurísticas anteriores numa heurística final, que atribui um peso relativo a cada uma dessas características fazendo o “trade-off” perfeito (ou quase perfeito) entre elas.

3.5.2 Forma de cálculo

Dado um nó X:

resultado = a * “heurística somatório das alturas para X” + b * “heurística linhas completas para X” + c * “heurística do numero de buracos para X” + d * “heurística da monotonia para X”

retorna o resultado

Nota: o cálculo das linhas completas é diferente caso estejamos a considerar 1 nível ou caso estejamos a considerar 2 e como tal a função que calcula esta heurística tem um argumento opcional para definir a função de cálculo de linhas.

Após correr o algoritmo genético explicado na secção 2.4 concluímos que as constantes associadas são a = 0.46069467, b = 0.56695324, c = 0.4459617 e d = 0.06362687.

Estas constantes surgiram de um “super-individuo” que surgiu na geração 10 e realizou em 10 jogos de 16 peças um total de 6100 pontos ou seja em média 610 pontos por jogo de 16 peças. Sendo que a média é de aproximadamente 490 pontos por jogo em todas as gerações.

4 Estudo Comparativo

4.1 Estudo Algoritmos de Procura

4.1.1 Critérios a analisar

Os critérios utilizados para comparar os algoritmos foram o tempo de cálculo e os pontos obtidos, visto que o objectivo é chegar a um algoritmo que faça uma procura que devolva de forma rápida e eficiente o caminho que leve a uma boa pontuação. Todos os outros critérios possíveis como nós expandidos, nós visitados etc.. reflectem-se de forma genérica em tempo de cálculo, já os critérios de qualidade reflectem-se em pontos, justificando assim a nossa escolha.

Também serão apresentados os resultados do algoritmo genético é importante relembrar que o critério de fitness para diferentes constantes foi apresentado anteriormente na definição do algoritmo.

4.1.2 Testes Efectuados

Efectuámos testes sobre a procura-pp, correndo o algoritmo passando-lhe problemas com os seguintes estados iniciais: 1º *tabuleiro vazio, pecas-por-colocar* (I I Z); 2º *tabuleiro vazio, pecas-por-colocar* (I Z I O); 3º *tabuleiro vazio, pecas-por-colocar* (L J Z S L Z S L J Z).

Para a procura-A* foram feitos testes para problemas cujo *custo-caminho* era a função *custo-oportunidade* e testes onde era a função *qualidade*, em ambos os testes a heurística era uma função nula. Para os testes com a função qualidade os estados iniciais foram os mesmos que para os testes da procura-pp, já para a função custo-oportunidade só se usaram os primeiros dois estados e em substituição do terceiro usou-se o estado: *tabuleiro vazio, pecas-por-colocar* (L J Z S L).

Como teste às procuras locais usaram-se sempre tabuleiros vazios e diversos conjuntos de peças. Tanto a procura local, como a de 2 *níveis* e a procura local *escolhe-k-best* foram comparadas com os mesmos testes. Os diferentes conjuntos de peças foram: 1º (I I Z), 2º (I Z I O), 3º (J O L O L O), 4º (L J Z S L Z S L J Z) utilizados para comparação de tempo e pontos. Para comparação só de pontos foram realizados testes sobre os seguintes conjuntos: 1º (T T I T I), 2º (O O T S O), 3º (O J O T T), 4º (O I L Z T), 5º (S S Z T O) todos de 5 peças.

4.1.3 Resultados Obtidos



Gráfico 4.1 – Resultados da procura-pp para 3, 4, 10 peças.

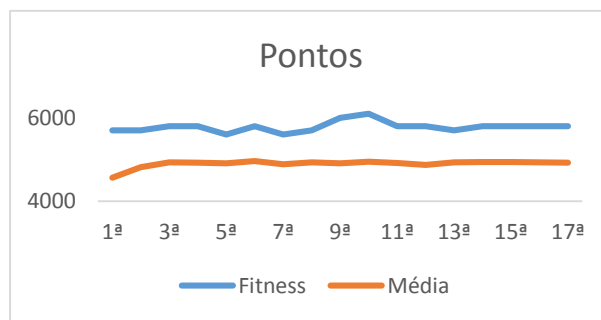


Gráfico 4.2 - Resultados da evolução do algoritmo genético relação à média e ao melhor indivíduo por geração

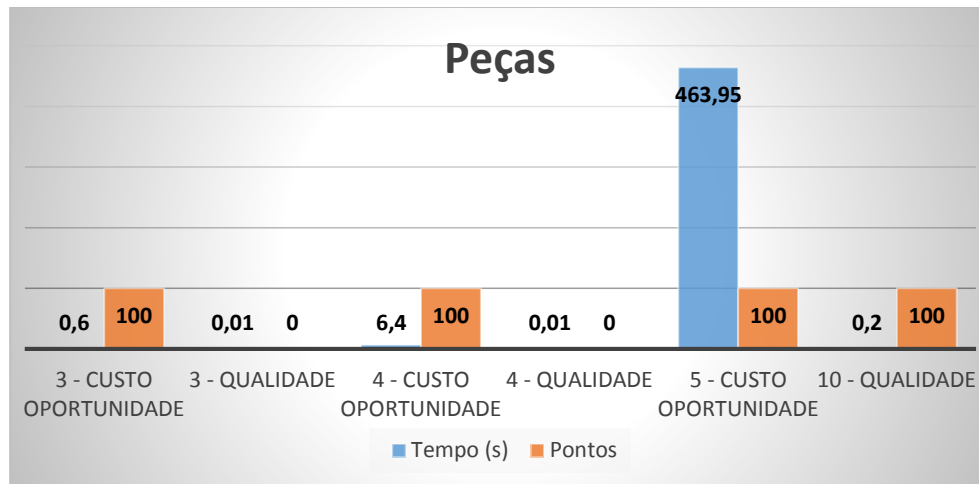


Gráfico 4.3 – Resultados dos diferentes testes à procura A* para 3,4,5 e 10 peças

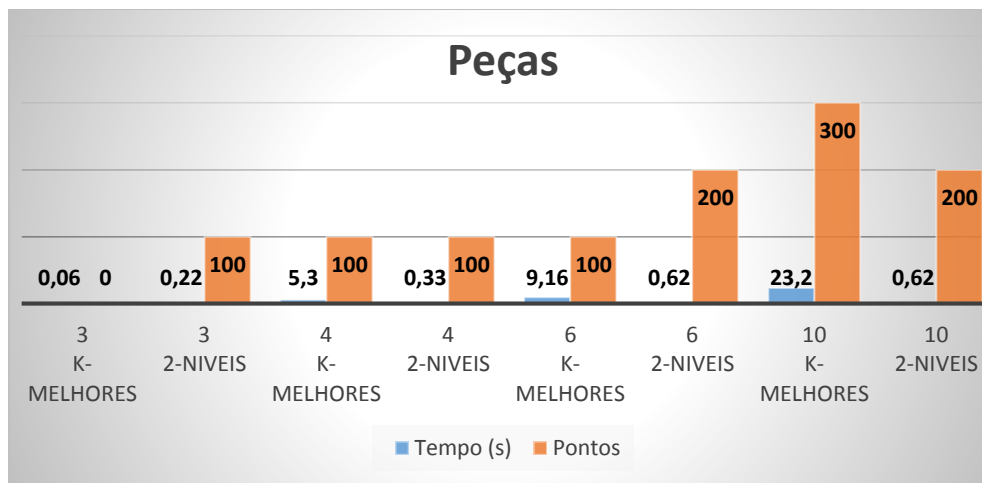


Gráfico 4.4 – Resultados da comparação das procuras locais

Tabuleiros	K-melhores (pontos)	2-níveis (pontos)
1º	100	100
2º	0	0
3º	100	100
4º	100	100
5º	0	0

Tabela 4.0 – Diferença de pontos para 5 tabuleiros de 5 peças entre as procuras locais

Não foram apresentados os resultados da procura local a 1 nível de profundidade visto que realizavam poucos pontos.

4.1.4 Comparação dos Resultados Obtidos

Após a realização dos testes conseguimos concluir que a procura-pp é bastante eficiente em termos de tempo mas não realiza pontos. Já a procura A*, dependendo da nossa função $f(n)$, consegue atingir altas pontuações. Este facto é visto pelos testes que usam a função *custo-caminho* como sendo o

custo-oportunidade e sem quaisquer heurísticas. Já os testes com a função *qualidade* demonstraram um comportamento igual à procura-pp.

Face às procuras locais, como podemos observar em termos de tempo, a escolhe-k-best é menos eficiente. No entanto se considerarmos 10 peças por colocar, consegue realizar mais pontos que a procura a 2 níveis. Este facto também é confirmado por testes adjacentes que foram realizados para 16 peças mas que decidimos não apresentar pela falta de espaço e visto que o projecto ia ser avaliado para um máximo de 6 peças. Um facto interessante e que nos ajudou na decisão para a escolha best é que para poucas peças (menos de 10) o comportamento da procura local a 2 níveis é melhor ou igual em termos de pontos.

Em suma a procura escolhe-K-best chega a maiores pontuações para um número de peças superior a 10 apesar de demorar no mínimo 20 segundos a calcular. Já a procura local a 2 níveis apresenta melhores resultados com menos de 10 peças e realiza o cálculo num máximo de 2 segundos. Para totais de 100 peças por exemplo esta procura calcula as acções em cerca de 16 segundos enquanto a escolhe-k-best tornasse uma má aposta devido ao tempo de cálculo necessário.

4.2 Estudo funções de custo/heurísticas

4.2.1 Critérios a analisar

Os critérios utilizados para o teste das diferentes heurísticas foi somente o número de pontos que eram realizados, visto que todas tinham um cálculo simples e foram testadas no algoritmo de procura local a dois níveis descrito na secção 2.3.2, tornando o critério referente ao tempo de resolução insignificante.

Desta forma o único critério que interessa é ver o comportamento face ao número de pontos obtidos por cada heurística e se estas levam a soluções sem ficarem presas em máximos locais, sendo um máximo local um estado onde não se conseguem colocar as peças todas sem perder o jogo.

4.2.2 Testes Efectuados

Os testes utilizados consistiram em correr a procura-best para um conjunto de peças por colocar que podiam ser de tamanho 16, 30 ou 100 num tabuleiro vazio e avaliar os pontos finais.

Os conjuntos de peças por colocar foram gerados aleatoriamente com a função *random-pecas* do ficheiro utils fornecido e são os seguintes:

Conjuntos de 16:

1º - (J I I J T O J Z Z Z I J Z I Z O)

2º - (J S T S T S J Z T Z T S I L S Z)

3º - (I Z I S Z T T J S S L S T O T T)

4º - (I Z Z J O I O O Z Z O O L T T O)

5º - (J I O T Z J L T O I I L O Z S Z)

Conjuntos de 30:

1º - (L Z O Z I Z J O T Z S Z J L Z Z J J Z O Z Z S J S J L J I O)

2º - (ISOSSJJITTOJZZZIJZIZOJSIZJSLOZ)

3º - (OTTJLJIZZOOSITOJOZLZZLTIILOJZ)

Conjunto de 100:

(ZSIISZTJIZTOIJOLIJZOSOSJSTZJOIJISOILOOTLSJZSJSSZZZZOSISSOSTOLT
IIJSOTTITLIJSSLOSLLTISOZTOTJJTZJJLSIIZS)

Para o mesmo número de peças como foi descrito anteriormente foram criados vários tabuleiros para eliminar ambiguidades.

4.2.3 Resultados Obtidos

Tabuleiro	1ª / 2ª / 3ª Tentativa (em pontos)		
1º	200	300	300
2º	500	500	100
3º	200	400	100
4º	300	0	400
5º	100	100	200

Tabela 4.1 – Testes à heurística Somatório de Alturas para tabuleiros de 16 peças

Tabuleiro	1ª / 2ª / 3ª Tentativa (em pontos)		
1º	800	800	900
2º	500	1200	900
3º	1300	800	900

Tabela 4.2 – Para tabuleiros de 30 peças

Tabuleiro	1ª Tentativa (em pontos)
1º	4100

Tabela 4.3 – Para tabuleiros de 100 peças

Tabuleiro	1ª / 2ª / 3ª Tentativa (em pontos)		
1º	100	100	100
2º	200	100	0
3º	100	0	0
4º	0	0	0
5º	0	0	100

Tabela 4.4 – Testes à heurística Linhas Completas para tabuleiros de 16 peças

A heurística para as linhas completas perdeu todos os jogos para 30 ou 100 peças, pelo que não as representamos.

Tabuleiro	1ª / 2ª / 3ª Tentativa (em pontos)		
1º	0	100	0
2º	0	300	Perdeu
3º	0	100	300
4º	0	400	0
5º	0	0	0

Tabela 4.5 – Testes à heurística Monotonia para tabuleiros de 16 peças

A heurística Monotonia perdeu o jogo de 100 peças, pelo que não o representámos.

Tabuleiro	1ª / 2ª / 3ª Tentativa (em pontos)		
1º	900	Perdeu	Perdeu
2º	800	Perdeu	Perdeu
3º	Perdeu	Perdeu	Perdeu

Tabela 4.6 – Para tabuleiros de 30 peças

Tabuleiro	1ª / 2ª / 3ª Tentativa (em pontos)		
1º	0	0	0
2º	0	300	100
3º	200	0	0
4º	200	100	100
5º	100	0	0

Tabela 4.7 – Testes à heurística Número de Buracos para tabuleiros de 16 peças

Tabuleiro	1ª / 2ª / 3ª Tentativa (em pontos)		
1º	200	Perdeu	30
2º	Perdeu	Perdeu	Perdeu
3º	100	Perdeu	Perdeu

Tabela 4.8 - Para tabuleiros de 30 peças

A heurística Número de Buracos perdeu o jogo de 100 peças, pelo que não o representámos. Face ao aspecto final dos tabuleiros de 16 peças podem ser consultadas as imagens da secção 3.

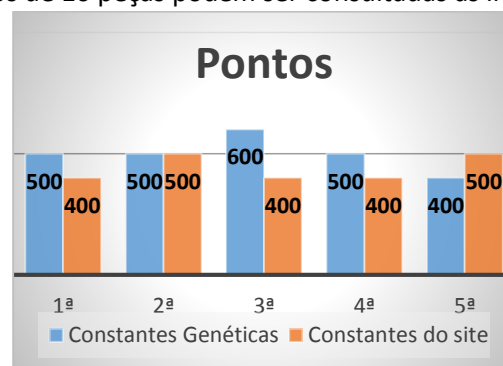


Gráfico 4.5 – Comparação das constantes calculadas pelo algoritmo genético com as constantes retiradas das nossas fontes

4.2.4 Comparação dos Resultados Obtidos

Através dos gráficos a cima podemos verificar que todas as heurísticas testadas individualmente fazem pontuações baixas, chegando mesmo a não fazer pontuação nenhuma e na grande maioria das vezes perdendo jogos com 30 peças.

No entanto todas elas nos permitem avaliar certas características específicas que são importantes para a qualidade de um boa jogada e que em alturas diferentes podem ser decisivas para o desenrolar do jogo. Esta suspeita é confirmada com os resultados apresentados no gráfico da meta-heurística que combina as 4 heurísticas através das constantes $a = 0.510066$, $b = 0.76066$, $c = 0.35663$ e $d = 0.184483$ retiradas do trabalho apresentado pelo site:

<https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/>

No entanto estas constantes foram calculadas com base no número de linhas eliminadas e específicas para o algoritmo apresentado no site. Para o nosso problema interessa mais a pontuação do que as linhas eliminadas, visto que fazer linhas 1 a 1 não traz assim grandes resultados. Como tal após o algoritmo genético cujos resultados se apresentam em cima representados podemos extrair as constantes $a = 0.46069467$, $b = 0.56695324$, $c = 0.4459617$ e $d = 0.06362687$ que apareceram na geração 10 e que o estudo comparativo com as constantes anteriormente testadas é apresentado na tabela X.

Apesar do algoritmo genético nos ter permitido chegar a um resultado satisfatório e que produz melhorias em relação as constantes utilizadas inicialmente o comportamento geral não foi o que o grupo estava a espera. Inicialmente pensávamos que á medida que eram criadas novas gerações melhores indivíduos fossem aparecendo, e no entanto isto não aconteceu. Tivemos a sorte de aparecer um “super-indivíduo” numa das gerações mas o que foi acontecendo é que a média foi estabilizando na casa dos 4900/5000 pontos por 10 jogos e o indivíduo best das gerações ia diminuindo o seu fitness. Dois factores que pensamos ter levado a este fenómeno é o erro na função mutação e a possibilidade de não ser possível melhorar mais a procura a 2 níveis.

4.3 Escolha da procura-best

Depois de testados os diferentes algoritmos e diferentes heurísticas escolhemos uma procura-best que utilizava a função *escolhe-melhor-neto* e caso o número de peças fosse impar a última era colocada através da função *escolhe-melhor-filho*. Como heurística foi utilizada a meta-heurística com as seguintes constantes: $a = 0.46069467$, $b = 0.56695324$, $c = 0.4459617$ e $d = 0.06362687$.

A escolha desta procura manteve-se fiel à ideia original. Após a leitura de alguns artigos em diversas fontes vimos que esta era a única forma de solucionar o problema de branching presente no Tetris e pensámos que o objectivo seria realizar uma procura que conseguisse resolver jogos com um número elevado de peças. Perto da entrega foi disponibilizada a informação que para a competição iriam ser avaliadas apenas 4 a 6 peças e foi com base nisso surgiu a necessidade de realizar a função *escolhe-K-melhores* que fosse até ao quarto nível. No entanto, como podemos observar, este algoritmo não tem um comportamento muito diferente nem compensa no tempo extra utilizado no cálculo, pelo que voltámos à ideia original realizando algumas optimizações na heurística número de linhas completas e criando um algoritmo genético que chegasse a melhores constantes.