Grupo Nº 29

Inteligência Artificial

1.º Semestre 2015/2016

IA-Tetris

Relatório de Projecto

Bruno Cardoso	72619
Francisco Calisto	70916
Lídia Freitas	78559

Índice

O repositório com a informação toda do projecto para consulta encontra-se em

https://github.com/lidiamcfreitas/AI-tetris

1 Implementação Tipo Tabuleiro e Funções do problema de Procura

1.1 Tipo Abstrato de Informação Tabuleiro

No projecto, optamos por utilizar um matriz bidimensional com dimensões 18x10 para representar o tipo tabuleiro. Esta opção foi tomada face aos tempos constantes de acesso a uma posição do tabuleiro, sendo assim possível verificar se determinada posição se encontra preenchida ou não em tempo constante, sem ser necessário percorrer listas. O facto de ser possível iterar sobre determinada coluna ou determinada linha também é claramente um ponto a favor da representação em matriz. O ponto fraco que encontramos para esta representação é o facto de quando uma linha é totalmente preenchida, ser necessário copiar as linhas acima uma a uma. No entanto, como existem muitos mais acessos a posições do tabuleiro do que vezes em que as linhas ficam totalmente preenchidas, optamos por utilizar a representação de matriz bidimensional.

No que toca a outros tipos de informação, outras alternativas que consideramos foram:

- representação apenas com um vector a 1 dimensão com 180 unidades. Isto trás a desvantagem de complicar o mapeamento quando queremos relacionar com o nosso tabuleiro de jogo, para alem de dificultar qualquer iteração a uma linha ou coluna, não trazendo assim nenhuma vantagem perante a nossa decisão.
- representação com listas, sendo o tabuleiro uma lista com 18 listas, cada uma com 10 elementos (correspondendo aos elementos da coluna). Esta representação apesar de ter um ponto a favor que é o facto de não ser necessário copiar e percorrer todas as posições das linhas acima de uma linha recentemente completa apresenta várias desvantagens, sendo a mais importante na nossa decisão o facto do tempo de acesso a uma posição do tabuleiro (operação mais vezes realizada) poder ser linha x coluna, enquanto que com uma matriz é constante.
- representação com uma lista que em cada elemento de uma lista tem uma matriz unidimensional que representa uma linha do tabuleiro. Esta implementação tem um aspecto positivo que é o facto de quando uma linha se encontrar completa ser apenas necessário retirar essa linha e inserir uma linha vazia no topo do tabuleiro, não sendo necessário copiar todas as posições acima mencionadas. No entanto, continua a ser necessário percorrer até 18 posições da lista para procurar determinada linha e, como acima visto, esta operação continua a ser muito mais frequente do que completar uma linha completa. Assim, mesmo com o tempo de acesso constante a uma determinada coluna dada uma linha e o facto de ser mais fácil calcular o tabuleiro resultante após uma linha completa, continuamos a achar que a matriz binária bidimensional a melhor estrutura para o nosso tabuleiro.

1.2 Implementação de funções do problema de procura

Para a correcta implementação do jogo de Tetris, foi necessária a implementação das seguintes funções:

- **qual-peca**: peca → lista de peças
- **accoes**: estado → lista de ações
- **solucao**: estado → lógico.

Para a implementação da função **accao** foi implementada uma função auxiliar, **qual-peca**, que passaremos a descrever.

qual-peca: Esta função recebe uma peça, e devolve, para aquela peça, uma lista com todas as configurações possíveis desta, que são indicadas no ficheiro *utils.lisp* pela ordem correcta de rotação.

solucao : Esta função recebe um estado, e determina se esse estado é solução de um problema. Para esta verificação, utilizamos a função **tabuleiro-topo-preenchido-p** para verificar que o topo do tabuleiro não está preenchido, e verificamos que já não existem mais peças para colocar através do atributo **pecas-por-colocar** da estrutura **estado**. Sendo estas duas verificações verdadeiras, concluímos que esse estado é solução, caso contrário retorna que o estado não é solução.

accoes: Esta função recebe um estado, e retorna uma lista com todas as ações válidas que podem ser efetuadas utilizando a próxima peça a ser colocada.
Para a verificação começamos por ver quais as configurações da próxima peça a ser colocada, utilizando a função auxiliar **qual-peca** acima descrita em combinação com a lista de próximas peças a colocar (pecas-por-colocar), para retirar a primeira peça da lista.

Para verificar se uma peça pode ser colocada no tabuleiro é avaliada a largura da peça de forma a que continue fisicamente possível colocar a peça no tabuleiro, continuando a retornar a ação mesmo que esta implique que o jogador perca o jogo, ou seja, que a linha do topo fique preenchida numa posição. Esta função chega ao fim depois de serem consideradas todas as configurações da próxima peça a ser colocada. É também verificado se o estado passado como argumento é um estado final e se for é retornado nil, caso contrário são retornadas as ações calculadas acima.

resultado: Esta função recebe um estado e uma ação, e devolve um novo estado que resulta de aplicar a ação recebida no estado original. Como estado original não pode ser alterado em situação alguma o estado original é copiado logo no inicio. Para o cálculo da altura mínima para colocar a peça recebida na ação são calculadas a máxima diferença entre a altura do tabuleiro e a altura do inverso da peça considerando a base como sendo zero, sendo este calculo linear na largura da peça.

De seguida a peça é colocada na coluna acima descoberta e é verificado se o topo se encontra preenchido e se sim retorna o estado inicial. De seguidas e apagadas o número de linhas completas e de acordo com esse número calculada o acréscimo dos pontos devido a essa ação.

Por fim, são somados os pontos à copia do estado inicial e é retirada a peça da lista de peças por colocar e inserida na lista de peças colocadas. É retornado o novo estado.

2 Implementação Algoritmos de Procura

2.1 Procura-pp

O algoritmo de procura em profundida primeiro é um algoritmo de procura que pode ser implementado em árvore em em grafo, com ligeiras alterações. Neste caso implementamos em árvore pois era o pedido no enunciado.

Numa procura em profundidade a fronteira comporta-se como um stack LIFO, em que o elemento gerado em ultimo será o primeiro a ser explorado na iteração seguinte. Para a implementação da stack LIFO foi utilizada uma lista de ações em que a ação mais recente é inserida sucessivamente no inicio da stack. Assim, a ação selecionada e retirada da stack é sempre a inicial, ou seja a mais recente.

Quando o resultado da dfs retorna impossível, ou seja, quando não é possível continuar a descer na árvore, o algoritmo retrocede e continua a procura por outro caminho. Se não for possível continuar o algoritmo então não existe solução e é retornado nil.

Para a resolução do problema é utilizada uma função auxiliar **dfs: estado -> lista acções** que implementa o algoritmo de procura em profundidade, podendo retornar *'impossível* se não existir solução possível para o problema, *nil* se o estado passado por argumento for solução do problema ou a lista de acções necessárias para a resolução da procura.

Nesta procura, é enviado o estado, e é iterada a lista de ações possíveis com esse estado. A cada iteração, verificamos se é solução, e recursivamente chamamos a função DFS até que tenhamos um resultado que satisfaça, ou cheguemos ao caso impossível.

2.2 Procura-A*

A Procura-A* caracteriza-se por tentar procurar primeiro um caminho que ache ser o mais eficiente no custo total do nó inicial até um nó objectivo. Este algoritmo é um algoritmo de procura que pode ser implementado em árvore em em grafo, com ligeiras alterações. No âmbito do projecto foi implementado em árvore, como pedido no enunciado.

Durante a execução da procura são armazenados **elementos** num min-heap. Estes elementos, instâncias da estrutura elemento possuem um valor que indica o valor do custo de caminho desde o estado inicial mais uma estimativa do custo para o estado final (heurística), um estado actual e uma lista de acções percorridas até então para chegar a esse elemento.

Para além das funções básicas para a implementação do heap criamos também as seguintes funções auxiliares:

- **add-to**, para inserir um elemento na lista de prioridade (min-heap)

- **pop-elem** para remover o elemento de menor valor da lista retornando-o
- **peek-elem**, retorna o primeiro elemento da lista de prioridade (heap), sem o eliminar.

Inicialmente a nossa implementação utilizava uma lista de elementos ordenada de forma ascendente como forma de guardar a fila-de-prioridade. No entanto esta implementação demonstrava-se muito morosa pois para a inserção de um elemento com elevado valor teria que percorrer a lista toda ou então reordenar a lista todas as vezes que um elemento era inserido.
Utilizando um min-heap o tempo de acesso ao elemento com menor valor continua a ser constante O(0) e no máximo o tempo de inserção como de remoção de um elemento é a altura da árvore em questão, ou seja O(log(n)) e não O(n), como na lista.

Na nossa implementação da Procura-A*, foi guardado o conjunto de nós por avaliar num min-heap, como acima mencionado, que inicialmente contém o nó inicial. Enquanto o heap não estiver vazio repete:

1. Retira o elemento com menor valor do heap e guarda em *top-elem*
2. Calcula a lista de acções possíveis a partir do *top-elem*
3. Para cada acção possível calcula o estado resultado de aplicar essa acção ao *top-elem* no *novo-succ*
4. Calcula o valor da estimativa de custo do caminho do estado inicial ao estado final passando pelo *novo-succ* tendo em conta o valor do custo real para chegar a este estado e a heurística.
5. Adiciona a nova ação à lista de ações necessárias para chegar do estado inicial ao *novo-succ*
6. adiciona o novo elemento com a informação anterior calculada ao min-heap de prioridades.

O valor da estimativa de custo de um caminho passando pelo estado n é dada por:

$$f(n) = g(n) + h(n)$$

Onde g(n) é o custo conhecido a partir do nó inicial, sendo este valor obtido através da função do problema *custo-caminho*. A função de estimativa h(n) é a heurística do custo para ir de n até qualquer nó objectivo. Para o algoritmo encontrar o caminho mais curto real, a função heurística deve ser admissível, o que significa que nunca superestima o custo real para chegar ao nó objetivo mais próximo. Por fim *f(n)* é a soma de *g(n)* com *h(n)*, denominada de função de custo que dá uma estimativa optimista do custo total. Este valor é utilizado para escolher o próximo nó a ser explorado sendo este escolhido se for o nó com menor valor de *f(n)*. Em caso de empate entre dois nós com igual valor de *f(n)* na lista deve ser escolhido o último a ter sido colocado no heap (fronteira).

2.3 Outros algoritmos

Não foram implementados quaisquer algoritmos adicionais para a descoberta de lista de acções que resulta na solução, sendo que o que foi utilizado foi a procura-A*. Contudo, foram implementados dois algoritmos, um random e outro genético para calcular os valores ideais para os pesos das heurísticas. As especificações e resultados destas implementações encontram-se numa parte mais à frente do relatório, onde são comparados os pesos das heurísticas utilizadas.

3 Funções Heurísticas

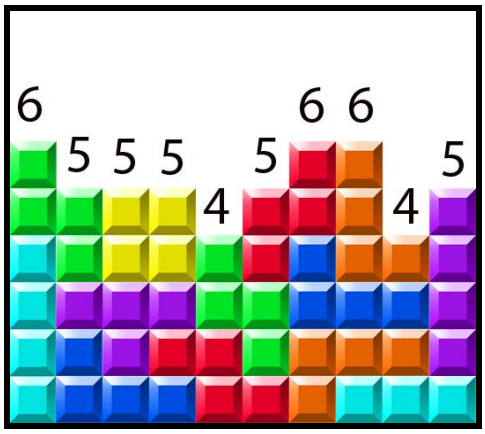
De forma a termos melhor noção de como maximizar o desempenho do nosso jogador de tetris inteligente, foi necessário termos uma ideia de como jogar o jogo e quais os objetivos na mente do jogador, para tal foi importante para nós jogar tetris para a elaboração das heurísticas.

Desta forma, observamos de que forma o jogador coloca as peças nas posições livres, e as escolhas que este faz em detrimento de outras também possíveis. Isto facilitou-nos a obtenção das heurísticas, sendo depois necessário apenas as afinações dos pesos de cada.
Desenvolvemos ao todo 5 Heurísticas, sendo uma a **heurística-geral**. Esta, a heurística geral destaca-se por ser uma combinação das restantes e sendo esta a heurística que pretendemos optimizar para o **procura-best**.

3.1 Heurística 1 - Altura Geral

3.1.1 Motivação

A primeira heurística a ser abordada é a heurística da altura geral, **heur-altura-geral**. Esta, como todas as outras, recebe um estado, e é invocada na heurística geral.
A ideia, ao jogar tetris, é que não tenhamos um jogo com uma altura demasiado alta, pois quanto maior a altura, maior é o risco de preenchermos a linha de topo, levando a que percamos o jogo.



Com esta heurística esperamos conseguir manter a altura do tabuleiro relativamente baixa, dando prioridade a estados com alturas menores.

3.1.2 Forma de Cálculo

Consideramos que esta heurística seria uma boa escolha para o nosso projecto, não só porque achamos que é bastante relevante para o problema mas também porque o seu cálculo é relativamente rápido.

O calculo da heurística consiste na soma das alturas de todas as colunas do tabuleiro. Sendo o objectivo posteriormente diminuir este valor.

Analisando o exemplo acima ilustrado, na secção 3.1.1:

O valor da **heur-altura-geral** é dado pela soma das alturas máximas de cada coluna, indicados na imagem. No exemplo, o valor de resultado é de 51 (= 6 + 5 + 5 + 5 + 4 + 5 + 6 + 6 + 4 + 5).

3.2 Heurística 2 - Buracos

3.2.1 Motivação

Ao jogarmos tetris reparamos que um dos maiores problemas para garantir a continuidade do jogo e a possibilidade de serem feitas novas linhas e encaixar as peças nos melhores locais é o facto de quando existe um buraco (quando uma posição se encontra não preenchida mas a imediatamente acima encontra-se preenchida) todas as posições da linha que contem o buraco ficam inalteradas até que este seja preenchido. Isto faz com que se forem preenchidas sucessivamente posições acima do buraco as linhas que as contém terem de ser eliminadas antes de qualquer outra linha abaixo da linha que contem o buraco onde existam posições preenchidas.

3.2.2 Forma de Cálculo

Para que possamos aplicar esta heurística, iteramos todas as posições do tabuleiro excepto a linha de topo, à procura de eventuais posições em que, estando livres, a posição na altura h+1 esteja preenchida, ocorrendo assim um **buraco**. Quando este facto ocorre é acrescentado o valor de 1 ao resultado, sendo este o número de buracos no tabuleiro de um dado estado.

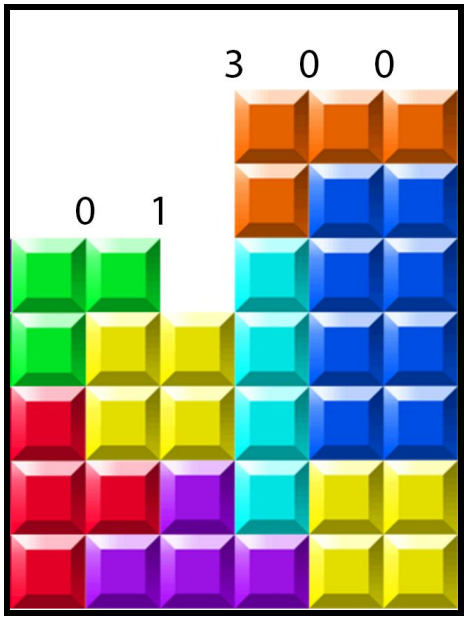
Analisando o exemplo acima ilustrado, na secção 3.1.1:

Como se pode verificar, na 1ª, 2ª e 3ª Coluna, existe uma buraco (em cada). Isto faz com que o jogo seja consideravelmente atrasado, pois é necessário eliminar todas as posições acima do buraco antes que seja possível retirá-lo, fazendo com que seja mais difícil decrementar a altura geral do tabuleiro, provando assim que existe uma relação entre as heurísticas.

3.3 Heurística 3 - Relevo

3.3.1 Motivação

Por análise de diversos jogos de tetris, não foram raras as vezes em que, ao jogar, formava-se um acumulado de peças em altura, como que uma torre, continuando a existir colunas sem posições preenchidas. Como descrito anteriormente, ter uma altura elevada de peças pode levar a que se atinja mais rapidamente o fim do jogo. Ao ter um relevo mais plano, ou seja, as diferenças absolutas de alturas entre colunas serem mais baixas facilita a colocação de peças (pois estas não têm relevos acentuados).



3.3.2 Forma de Cálculo

Para calcularmos esta heurística, temos que percorrer o nosso tabuleiro, coluna a coluna, calculando as diferenças absolutas de alturas de colunas. De seguida soma-se essa diferença absoluta ao resultado, que no final terá a informação do relevo total do tabuleiro.

3.4 Heurística 4 - Custo de Oportunidade

3.4.1 Motivação

Outra heurística que consideramos e que nos era pedida para implementar no relatório é a heurística custo de oportunidade. Uma forma de analisá-la é vendo que podemos potencialmente ganhar um determinado valor, e que o custo é dado pelo facto de não conseguirmos ter aproveitado ao máximo a oportunidade. Assim sendo o custo de oportunidade pode ser calculado como a diferença entre o máximo possível e o efetivamente conseguido ao colocar uma peça.

Portanto esta heurística, dado um estado, devolve o custo de oportunidade de todas as ações realizadas até ao momento, assumindo que é sempre possível fazer o máximo de pontos por cada peça colocada.

Assim sendo, para tentar maximizar o número de pontos num jogo, deve-se tentar minimar o custo de oportunidade.

3.4.2 Forma de Cálculo

O cálculo desta heurística começa por receber um determinado estado. De seguida percorre cada peça existente na lista de peças-colocadas onde é feita uma contagem dos pontos máximos que seriam possíveis de obter caso cada peça eliminasse todas as linhas que o seu formato possibilita. Recorremos assim à função auxiliar **calcula-pontos-por-peca** de forma a que para cada peça, tenhamos o seu valor máximo. No final, retiramos a esse acumulador, o número de pontos conseguidos até ao momento, que já vinham no estado, ficando assim com o custo de oportunidade.

3.5 Heurística 5 -Heurística Geral

3.5.1 Motivação

Após a análise do problema concluímos que a melhor forma de o resolver seria uma combinação das heurísticas anteriormente mencionadas, assim criamos a heurística geral que combina todas as heurísticas analisadas procurando arranjar a melhor relação entre elas, multiplicando cada uma pelo peso que estas têm na resolução do problema.

3.5.2 Forma de Cálculo

Esta nossa ultima heurística e trata-se da combinação das restantes previamente descritas como acima mencionado, sendo esta a heurística que pretendemos optimizar para o **procura-best**.

Esta envia o estado que recebe como argumento para cada uma das outras heurísticas, e multiplica-as pelo valor de peso respectivo.

4 Estudo Comparativo

4.1 Estudo Algoritmos de Procura

4.1.1 Critérios a analisar

Para análise dos algoritmos de procura A* e Profundidade Primeiro, utilizamos 2 métricas para o cálculo do desempenho.

Como primeiro critério, verificamos se o tempo de execução ultrapassa os 20 segundos. Se assim for, é **atribuída uma pontuação de -500 a um teste que atingiu o limite de tempo**, pois é consideramos de tempo elevado dadas as restrições temporais do problema e permite-nos continuar a comparar resultados dando uma penalização considerável à procura.

Como segundo critério e o principal, obtemos os pontos conseguidos em cada execução completa, tentando maximizar os resultados para tabuleiros predefinidos.

4.1.2 Testes Efetuados

De forma a testarmos os algoritmos de procura, tivemos que encontrar vários casos de uso, e ainda, utilizar elementos, como tabuleiros e conjuntos de peças diferentes em número significativo, de forma a que os resultados sejam expressivos.

Atendendo às nossas limitações de tempo e recursos computacionais, testes exaustivos não foram possíveis, mas tentamos testar com estruturas e intervalos de valores consideráveis para conseguirmos tirar conclusões viáveis.

Para a procura PP aguardamos apenas o seu valor de output, dado um problema. Tivemos sempre em consideração o seu tempo limite de execução que estabelecemos como sendo de 20 segundos. Assim sendo obtivemos sempre dois resultados possíveis: ou a procura retorna impossível pois o tempo acabou, ou a procura retorna o valor de pontos que a sua solução consegue atingir.

Para uniformizarmos os testes, geraram-se 10 tabuleiros de forma aleatória, recorrendo à função *tabuleiro-random*, fornecida no *utils.lisp*, e guardados num ficheiro, um por linha. Esta situação é para garantir que, para as 2 procuras a serem testadas, os tabuleiros apesar de gerados aleatoriamente, são os mesmo em cada caso. A execução dos testes desta forma foi feita para nos permitir conseguir comparar resultados entre procuras.

No que toca às peças, são igualmente testados os mesmos conjuntos de peças para cada procura. Para tal foram feitos 4 conjuntos de peças, que incluem conjuntos com 5, 7, 8 e 29 peças nos testes realizados entre a **procura-pp** e a **procura-A*.**

Para a comparação da procura-pp e da procura-A* foram realizados 160 testes:

- para cada um dos **10 tabuleiros**
- foram utilizados **4 conjuntos de peças**
- com **3 algoritmos** diferentes como testados nos testes públicos:
    - a procura-pp
    - o A* com custo de caminho igual à função qualidade e heurística = 0
    - o A* com custo de caminho igual à função de custo de oportunidade e heurística = 0

De forma a automatizar todo este processo, foi criado um *script* (**ver Anexo 4.6.2**) que de forma automática faz com que as procuras retornem os valores de resultado obtidos.

4.1.3 Resultados Obtidos na comparação da procura-pp e A*

Dos nossos 120 testes executados, obtivemos os seguinte resultados:

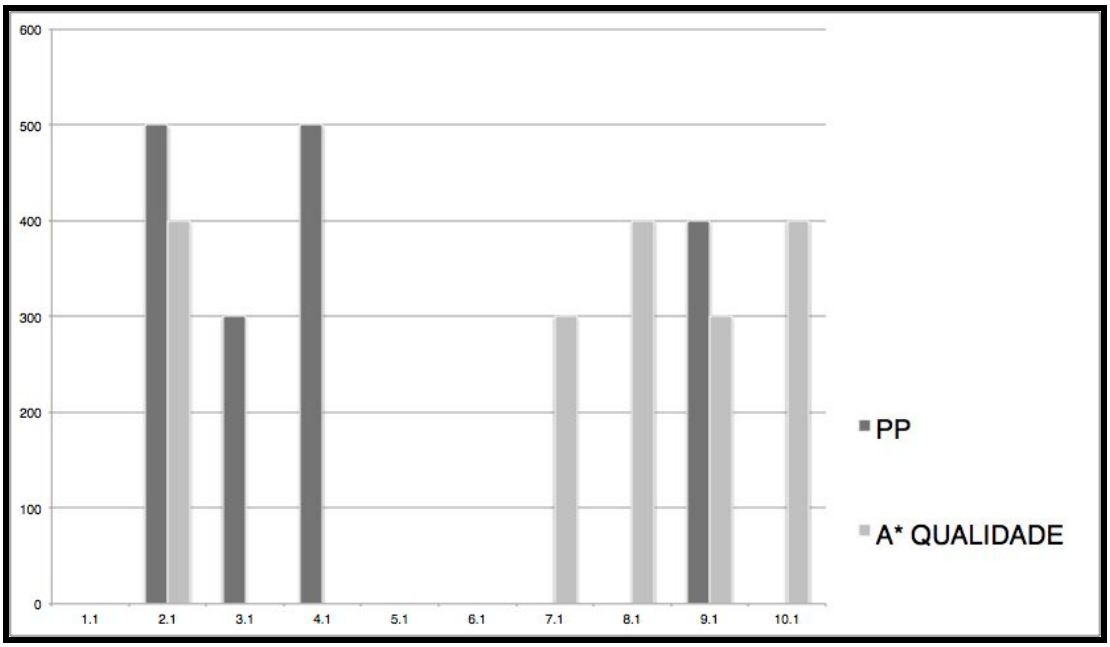


Gráfico 1

Este gráfico ilustra os resultados obtidos sempre que o limite de tempo não foi excedido. Por motivos de legibilidade, e derivado de termos dado valor -500 a todos os testes que esgotaram o tempo, esses não foram incluídos no gráfico acima descrito. Por esta razão, nas colunas que estão sem valor, não implica que dai tenha resultado valor 0 de pontos.

Como pudemos observar e analisando em conjunto tanto o Gráfico 1 em cima como também o Anexo 2 no final deste relatório, concluímos que nos testes 1.1, 5.1 e 6.1 não houve de facto nenhum resultado viável em ambas as procuras.

Observando os testes 2.1 e 9.1, nos casos em que as procuras Procura-PP e Procura-A* Qualidade ambas terminam com um valor positivo, a Procura-PP consegue sempre um melhor resultado. Sendo poucos os testes em que conseguimos resultados de ambas as procuras, não é uma facto forte que se possa regra.

As conclusões que retiramos dos testes é que apesar da Procura-A* Qualidade utilizar a heurística de qualidade para custo caminho apresentar resultados razoáveis mais vezes do que a procura-pp (5 contra 4) não apresenta no geral bons resultados. No entanto é na Procura-PP que obtemos os melhores valores, quando ambas terminam. Concluímos assim que nenhuma das procuras que analisamos é ideal, visto que apenas obtivemos resultados positivos para a procura-pp em 4 casos e a

A* com custo de caminho igual à função qualidade em 5 casos. Sendo que na procura A* com custo de caminho igual à função de custo de oportunidade não obtivemos resultados positivos.

4.1.4 Comparação dos Resultados Obtidos

Após 120 testes, obtivemos números expressivos das melhores opções para as procuras. Em primeira análise, e para os nossos casos de teste, descartamos imediatamente a procura A* tendo custo-caminho a heurística custo-oportunidade, pois esta ultima excedeu o nosso time-limit de 20 segundos para os 10 tabuleiros, com 4 configurações de peças diferentes.

Já a procura PP e a procura A* com a função de qualidade para o custo de caminho obtiveram melhores resultados. Numa primeira observação, concluímos que, para taxas de sucesso superiores, é preferível que se tenha um maior numero de peças. Nestas procuras, o uso de 29 peças fica bastante próximo do tempo limite dos 20 segundos, mas conseguem-se boas pontuações na maior parte das vezes nestes testes. Quando o numero de peças é inferior às 9 peças, a pontuação acaba por ser zero.

Nos casos em que obtivemos pontos, concluiu-se que a procura Profundidade Primeiro pontuou -1300 Pontos, enquanto que a procura A* pontuou -700 pontos, tendo em consideração que não conseguir acabar um teste por excesso de tempo resulta numa penalização de -500 pontos.

De seguida, apresentam-se os resultados das 10 procuras PP e das 10 procuras A* utilizando a heurística de qualidade para o custo-caminho.

Procuras	1.1	2.1	3.1	4.1	5.1	6.1	7.1	8.1	9.1	10.1	SUM
PP	**-500**	500	300	500	**-500**	**-500**	**-500**	**-500**	400	**-500**	**-1300**
A*Qualidade	**-500**	400	**-500**	**-500**	**-500**	**-500**	300	400	300	400	**-700**

4.2 Estudo funções de custo/heurísticas

4.2.1 Critérios a analisar

Das diversas heurísticas desenvolvidas, a nossa intuição indicou-nos que todas são essenciais para o sucesso do nosso jogo de Tetris, contudo, decidimos que fosse o jogo a decidir se tal seria verdade calculando os melhores pesos para a ponderação das heurísticas utilizando um algoritmo random e um genético.

Esta escolha deriva do facto de ser essencial obter um valor de pesos ideal, que cubra ao máximo os diversos tipo de tabuleiro e combinações de peças. Para isto, recorremos á nossa procura procura-best, a 3 tipos de tabuleiros aleatórios, com existiram 3 tabuleiros para cada tipo (total de 9). Para isto, foram feitos testes em número que seja suficientemente expressivo, em que os valores dos pesos das heurísticas foram gerados aleatoriamente numa fase inicial, até que se obtivesse uma

aproximação dos melhores valores. De seguida os 5 melhores valores foram utilizados para um algoritmo genético adaptado de forma a aprimorar os valores dos pesos das heurísticas.

4.3 Algoritmo Random

Apesar de um algoritmo random muitas vezes ter uma conotação negativa, em muitos casos os resultados deste demonstram-se bastante úteis e por vezes melhores do que uma implementação que se pensava ser boa. Por isso decidimos implementar um algoritmo random de procura de bons pesos para as heurísticas.

4.3.1 Testes Efetuados

Foram efetuados 41 testes, com todos os valores de pesos das heurísticas gerados aleatoriamente entre 0 e 1. Como os **valores das heurísticas são normalizados** na heurística geral, a multiplicação por este valor torna-se possível para futuras comparações. Como em todos os testes efetuados neste relatório, o tempo de execução de cada teste foi limitado a 20 segundos, para permitir uma comparação entre heurísticas foi atribuído um valor de -500 aos testes que ultrapassassem o limite temporal.

Para cada conjunto de pesos random de heurísticas são corridos os testes com a procura best em 9 tabuleiros diferentes e somados os resultados de cada teste. Para comparar pesos de heurísticas são analisados os valores das somas dos resultados da procura. **[Ver Anexo 1]**

4.3.2 Resultados Obtidos

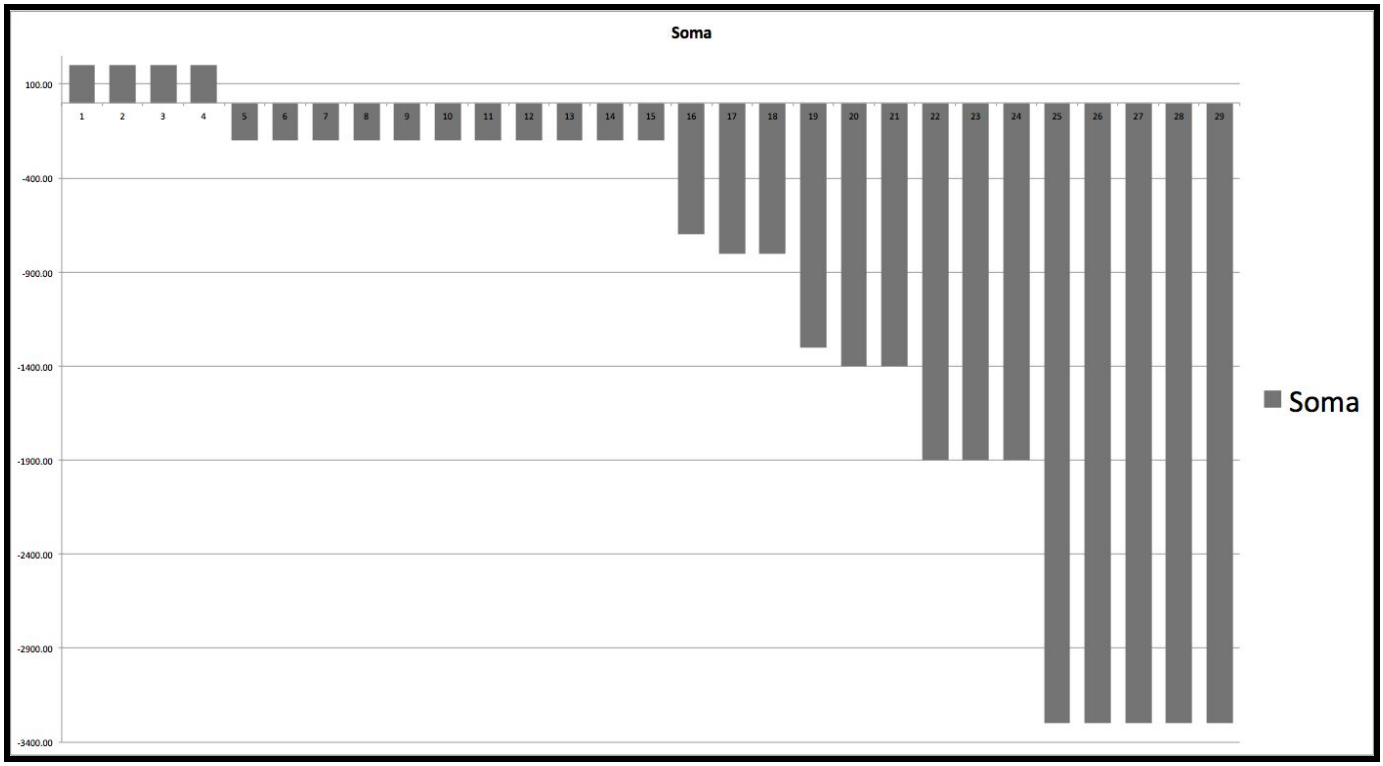


Gráfico 2 **[Ver Anexo 1]**

Este gráfico ilustra a soma por teste, do valor de pontos, já por ordem da soma. Como seria de esperar de um algoritmo random, as combinações de pesos das heurísticas fazem com que na maior parte dos tabuleiros a procura não termine, recebendo um valor de -500. No entanto, analisando com detalhe o **Anexo 1** conclui-se que apesar de existirem muitas combinações más, o algoritmo random conseguiu chegar a algumas combinações relevantes de pesos, nomeadamente 4 com valor de 200.

Note-se que foram feitos 41 testes, contudo, e dado Soma estar ordenado por valores positivos, os últimos 11 testes foram retirados da tabela, pois o seu valor de soma era demasiado reduzido (na ordem dos -4000), ficando excluídos do gráfico por motivos de legibilidade.

4.3.3 Comparação dos Resultados Obtidos

Dos 41 testes obtidos, verificou-se que apenas 4 totalizaram valores de soma positivos(de 1 a 4), e apenas 11 tiveram valores de soma negativos mas pouco acentuados.

Por análise dos valores na tabela em anexo(tabela 4.6.1, Anexo 1), um dos nossos tabuleiros fez com que nenhum valor de heurística fosse um bom valor, pois excedeu sempre os 20 segundos. Em contraste com este, o nosso tabuleiro 8 foi também um caso especial, pois nunca excedeu o tempo, mas também não obteve nenhum valor positivo, sendo sempre 0, concluindo que pelas heurísticas analisadas este não deve ser possível.

A partir destes resultados tomamos como base os 5 melhores pesos das heurísticas, de forma a inicializar o algoritmo de procura genética já com um bom conjunto.

4.4 Algoritmo Procura Genético Adaptado

Devido aos pesos das heurísticas serem dependentes dos valores das outras heurísticas e ser importante a relação entre pesos, achamos que seria uma boa ideia implementar um algoritmo genético de forma a tentar melhorar as boas características de um conjunto de pesos cruzando conjuntos que sabemos serem bons.

Para tal utilizamos um algoritmo genético bastante simplificado, com algumas alterações: este não realiza uma mutação num peso de uma heurística com uma dada probabilidade, mas sim acrescenta um novo conjunto totalmente aleatório para ser testado e o cruzamento de informação é feito aleatoriamente entre um conjunto de 5 melhores.

Uma implementação melhorada destes aspectos seria algo que gostaríamos de implementar no futuro.

4.4.1 Testes Efetuados

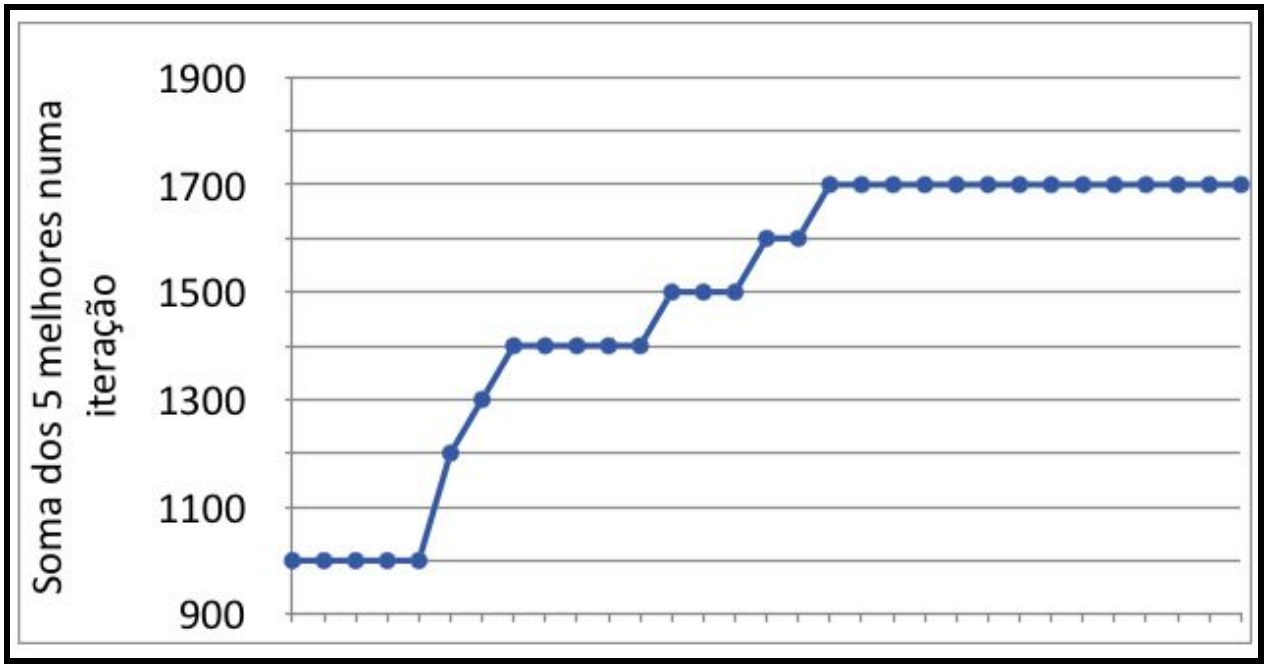
Para o algoritmo de procura genético foram realizadas 30 iterações do algoritmo, tendo sido inicializado o conjunto de 5 sets de valores de pesos para as heurísticas com os melhores valores obtidos no algoritmo random.

Cada iteração do algoritmo de procura genético consiste em:

- Gerar 9 conjuntos realizando cruzamentos aleatórios com 5 melhores
- Gerar um conjunto de pesos aleatório
    - Para cada um dos 10 conjuntos, 9 filhos + 1 random:
        - Calcular a soma dos valores dos resultados para cada um dos 10 tabuleiros
- Calcular quais os novos 5 melhores com base nos valores dos 9 filhos, 1 random e nos 5 melhores originais
- Guardar esses novos 5 melhores e repetir com estes.

Para a análise do código do algoritmo: **Ver Anexo 6**

4.4.2 Resultados Obtidos



Como acima mencionado, para cada conjunto de pesos foi realizada a soma dos valores dos resultados para cada tabuleiro. Agora, para cada iteração temos 5 melhores conjuntos, e foi realizada a soma desses valores para cada conjunto. Esse valor é indicado no gráfico acima no eixo vertical. No eixo horizontal estão dispostas as 30 iterações do algoritmo genético por ordem de execução.

Para um estudo mais detalhado dos resultados obtidos por favor veja o **Anexo 5.**

Como seria de esperar, num algoritmo genético começando com um determinado conjunto e realizando o cruzamento das características (pesos) os resultados tendem a ser bastante bons.

Nos nossos resultados obtidos, as primeiras 5 iterações mantiveram o mesmo resultado total que no algoritmo random, nas 3 iterações seguintes este resultado subiu sempre: na primeira um conjunto de pesos das heurísticas teve valor de soma de 400, na segunda iteração o 400 manteu-se e foi gerado outro conjunto com valor de 300, na terceira iteração já havia 2 conjuntos de 300 e um de 400. De seguida os conjuntos de pesos que tinham valores de 200 foram sendo substituidos por outros com valores de 300, gradualmente e por fim terminou-se o conjunto de 5 melhores com os pesos totais de: 400, 400, 300, 300, 300. Dando assim uma final de 340 pontos no total dos 9 tabuleiros. Como se pode verificar, o melhor resultado da procura genética (400) é bastante superior ao melhor resultado da procura random (200), como seria de esperar.

4.4.3 Comparação dos Resultados Obtidos

Tendo em conta que inicializamos o nosso algoritmo de procura genética com os melhores valores de peso das heurísticas, conseguidos no estudo anterior, o nosso algoritmo de procura genética inicializou com 1000 de valor de soma das 5 melhores iterações e foi sempre aumentando esse valor ao longo dos testes.

No **Anexo 5** encontram-se as tabelas resultantes das procuras efetuadas. Estas incluem uma coluna a indicar o numero de cada teste dos 5 melhores, pesos para as heurísticas respectivas, valores resultantes e o somatório por grupo de 5 melhores.

É interessante a análise detalhada dos resultados e perceber-se que a cada iteração o algoritmo percebe um pouco melhor como resolver dado tabuleiro misturando os conjuntos anteriores.

Esta análise faz também chegar à conclusão que o algoritmo se adapta aos testes a que está submetido. Este facto pode tornar-se um problema, e poderá ser o nosso caso visto que apesar de termos tentado sido bastante exaustivos nos testes apenas estamos a considerar 9 tabuleiros. Idealmente deveriam testados muito mais, de forma a que as heurísticas não fiquem demasiado "presas" a um conjunto de testes.

Os pesos das heurísticas que deram os melhores resultados (400) são:

Heur. Buracos	**Heur.Relevo**	**Heur. Altura Geral**	**Heur. Custo-Oportunidade**
0,17679017	0,99800133	0,47329903	0,36889546
0,25373280	0,93107579	0,27038572	0,33585559

Como se pode verificar no conjunto de pesos eleito o algoritmo genético este decidiu que a heurística relevo é a que tem mais influência na resolução dos tabuleiros, e muito pouca importância para a heurística dos buracos.

4.5 Escolha da procura-best

Na nossa implementação do *procura-best* este recebe um tabuleiro e uma lista de peças por colocar, criando assim um novo problema de tetris, onde este juntamente com a *heurística-geral* e o *procura-A** tentam resolver o problema. A *heurística-geral*, heurística usada para combinar as restantes heurísticas para ser então usada nesta procura, através da aplicação da função *procura-A** onde o primeiro argumento é o nosso problema e o segundo argumento a nossa *heurística-geral*.

Voltando agora a falar da *heurística-geral*, esta é a combinação das **Heurísticas Buracos**, **Relevo**, **Altura Geral** e **Custo Oportunidade**, onde todas as heurísticas são divididas pelo valor máximo que podem atingir de forma a ficarem normalizadas de 0 a 1. Estas são posteriormente multiplicadas pelos pesos das heurísticas, que de acordo com a nossa pesquisa utilizando o algoritmo genético, que atingiram os melhor resultados são:

- 0,17679017 para a **Heurística Buracos**
- 0,99800133 para a **Heurística de Relevo**
- 0,47329903 para a **Heurística de Altura Geral**
- 0,36889546 para o **Custo Oportunidade**

Durante a realização do relatório optamos por os pesos acima referidos por considerarmos que são os melhores para os casos testados, no entanto estes são diferentes dos submetidos no mooshak.

A razão para tal facto é porque apesar do esforço para encontrar os melhores pesos das heurísticas, como não realizamos os testes de procura de melhores pesos com o tabuleiro do teste 25 os resultados das heurísticas estão optimizados para os 9 tabuleiros para os quais foram testados e portanto não passam ao teste 25 do mooshak.

4.6 Anexos

Os *scripts* foram realizados parte em lisp e parte em python, devido à necessidade de correr comandos na consola para controlar o limite de tempo de execução das procuras. Assim foi possível utilizar o comando *gtimeout* de forma a utilizarmos a informação de tempo excedido como resultado nos algoritmos. Como não encontramos uma maneira de correr comandos do terminal em lisp e acedermos ao resultados destes, assim como não abortar o código quando ocorre um erro, decidimos utilizar um script de python que executa o script de lisp.

4.6.1 Anexo 1 - Algoritmo Random

Heuristica 1	Heuristica 2	Heuristica 3	Heuristica 4	tab 0	tab 1	tab 2	tab 3	tab 4	tab 5	tab 6	tab 7	tab 8	Soma
0.06252293039487711	0.6169685981711657	0.5869438219752874	0.04369354680040316	0	100	100	200	100	200	0	-500	0	200
0.9128518893940584	0.4226895567776138	0.9134927770474299	0.06251677931489352	0	100	100	200	100	200	0	-500	0	200
0.25373280159762823	0.9310757885884794	0.7614826005183634	0.06311617522444746	0	100	100	200	100	200	0	-500	0	200
0.016441216301480432	0.3383680738360155	0.2703857247354139	0.06780098767615228	0	100	100	200	100	200	0	-500	0	200
0.4646313036167071	0.6984126477276682	0.7304445618597235	0.3358555934538632	-500	100	100	200	200	200	0	-500	0	-200
0.6320113610652446	0.23774505062086482	0.3896260285128641	0.31688890332578057	-500	100	100	200	200	200	0	-500	0	-200
0.6995339334009267	0.04081511646811198	0.579972588121868	0.4594984488448236	-500	100	100	200	200	200	0	-500	0	-200
0.03601277930365099	0.5660937752032108	0.30615363227222825	0.32329231099725986	-500	100	100	200	200	200	0	-500	0	-200
0.15679065865869046	0.7933491167072618	0.607629672311599	0.3346972985946902	-500	100	100	200	200	200	0	-500	0	-200
0.6822749866699295	0.7978098079649162	0.8182393248895963	0.37442660945123796	-500	100	100	200	200	200	0	-500	0	-200
0.7847512943293362	0.2961250159293226	0.229809201767729	0.17583823673610866	-500	100	100	200	200	200	0	-500	0	-200
0.077715906274615	0.8100963642026789	0.20774106813642967	0.33360802637993925	-500	100	100	200	200	200	0	-500	0	-200
0.7778964511214592	0.8483888595469724	0.9513170196249137	0.304732824010407	-500	100	100	200	200	200	0	-500	0	-200
0.09466569774357736	0.6389455501248846	0.8973317442766121	0.19238553180294982	-500	100	100	200	200	200	0	-500	0	-200
0.19177002985341474	0.8604337408436806	0.8327473820674214	0.4829112438312335	-500	100	100	200	200	200	0	-500	0	-200
0.18999906847437542	0.21514283800588863	0.12839755247735996	0.22955252894936085	-500	100	100	200	200	200	-500	-500	0	-700
0.3004545202440063	0.3134478279297872	0.9469545628226209	0.4561657774418265	-500	-500	100	200	200	200	0	-500	0	-800
0.5161021176561562	0.8384627463094789	0.4535529809962482	0.6722389437582347	-500	100	-500	200	200	200	0	-500	0	-800
0.05831122775889375	0.05395002651089409	0.3775524391040168	0.29332694144167826	-500	-500	100	200	200	200	-500	-500	0	-1300
0.2088459782826232	0.8772861115378444	0.8606725017602536	0.8437547099102969	-500	-500	-500	200	200	200	0	-500	0	-1400
0.5740722631374976	0.2815017961816483	0.4143334789566411	0.5323338979302508	-500	-500	-500	200	200	200	0	-500	0	-1400
0.22330209249732758	0.4032844984056917	0.8224978401571105	0.5647123156530872	-500	-500	-500	200	200	200	-500	-500	0	-1900
0.4781997838068113	0.35827279153694214	0.6989538332706697	0.6299694356017008	-500	-500	-500	200	200	200	-500	-500	0	-1900
0.4740112619675746	0.7290605057847331	0.9146229512048311	0.8183764031471531	-500	-500	-500	200	200	200	-500	-500	0	-1900
0.6967214914316192	0.4590693755810894	0.8810443170478817	0.9046305523290941	-500	-500	-500	-500	200	-500	-500	-500	0	-3300
0.8496356605195472	0.6812744761426828	0.841031475425354	0.9722137578542082	-500	-500	-500	-500	200	-500	-500	-500	0	-3300
0.9914447604171065	0.021070015218511373	0.5993431284458963	0.668349481150732	-500	-500	-500	-500	200	-500	-500	-500	0	-3300
0.1453216259328356	0.2718517262003526	0.4942729266322392	0.607748928962906	-500	-500	-500	-500	200	-500	-500	-500	0	-3300
0.5387259309592755	0.11282761138258368	0.5769604485745694	0.7129771534538232	-500	-500	-500	-500	200	-500	-500	-500	0	-3300
0.9690275953213713	0.8367419719601428	0.01028235550654455	0.7247168121445109	-500	-500	-500	-500	-500	-500	0	-500	0	-3500
0.03756083107618147	0.48359416947564826	0.7362826615434248	0.9692574232437191	-500	-500	-500	-500	-500	-500	-500	-500	0	-4000
0.5612881766174663	0.7255099640492566	0.012333374576918299	0.984933997977163	-500	-500	-500	-500	-500	-500	-500	-500	0	-4000
0.12143557785056447	0.4766113921337789	0.18463990994592672	0.9094467398639591	-500	-500	-500	-500	-500	-500	-500	-500	0	-4000
0.5346274569593081	0.3570919295116384	0.3771748133595221	0.7978735364135261	-500	-500	-500	-500	-500	-500	-500	-500	0	-4000
0.359672314392648	0.1728823000287435	0.0779307149028522	0.745653093489562	-500	-500	-500	-500	-500	-500	-500	-500	0	-4000
0.6364783824210489	0.29399070985715203	0.7160097862277969	0.9353321718829716	-500	-500	-500	-500	-500	-500	-500	-500	0	-4000
0.07489068414269473	0.29960733576968757	0.4351777666120097	0.843199691026713	-500	-500	-500	-500	-500	-500	-500	-500	0	-4000
0.1680004135813432	0.16944520379829486	0.16091260018613573	0.7320628456398167	-500	-500	-500	-500	-500	-500	-500	-500	0	-4000
0.16173112905686615	0.23001934361745702	0.4524015841603638	0.6160834595559418	-500	-500	-500	-500	-500	-500	-500	-500	0	-4000
0.5710325583338719	0.44555386837258004	0.8237999740409542	0.984751412744559	-500	-500	-500	-500	-500	-500	-500	-500	0	-4000
0.42034389103294767	0.3051556434634062	0.6821280551757313	0.8682752849736677	-500	-500	-500	-500	-500	-500	-500	-500	0	-4000

4.6.2 Anexo 2 - Análise Procura-PP e A*

	1.1	1.2	1.3	1.4	2.1	2.2	2.3	2.4	3.1	3.2	3.3	3.4	4.1	4.2	4.3	4.4	5.1	5.2	5.3	5.4
PP	-500	0	0	0	500	0	0	0	300	0	0	0	500	0	0	0	-500	0	0	0
A* QUALIDADE	-500	0	0	0	400	0	0	0	-500	0	0	0	-500	0	0	0	-500	0	0	0
A* CUSTO-OPORT	-500	-500	-500	-500	-500	-500	-500	-500	-500	-500	-500	-500	-500	-500	-500	-500	-500	-500	-500	-500

	6.1	6.2	6.3	6.4	7.1	7.2	7.3	7.4	8.1	8.2	8.3	8.4	9.1	9.2	9.3	9.4	10.1	10.2	10.3	10.4	
PP	-500	0	0	0	-500	0	0	0	-500	0	0	0	400	0	0	0	-500	0	0	0	-1300
A* QUALIDADE	-500	0	0	0	300	0	0	0	400	0	0	0	300	0	0	0	400	0	0	0	-700
A* CUSTO-OPORT	-500	-500	-500	-500	-500	-500	-500	-500	-500	-500	-500	-500	-500	-500	-500	-500	-500	-500	-500	-500	-20000

4.6.3 Anexo 3 - lispscript0.lisp e script0.py

lispscript0.lisp

```
#! /usr/local/bin/clisp

(setf valor0 (nth 0 *args*))

(load "entregue.lisp")
```

script0.py

```
#!/usr/bin/env python3

import random
import subprocess
import glob
random.seed()

heur_random_values = [random.random() for i in range(4)]

print("random values:" , heur_random_values)

# vais buscar uma lista de todos os testes do directorio
ficheiros = glob.glob("./xx*")

for ficheiro in ficheiros:
   cmd = "gtimeout 20s ./lispscript0.lisp " + ficheiro

# resultado do comando na consola, que inclui os prints do script em lisp
cmd_result = subprocess.check_output(cmd, shell=True)

print("running command...    ", cmd)
print("result: ", cmd_result)
result = [int(s) for s in cmd_result.split() if s.isdigit()]
print  (result) # resultado do teste
```

4.6.4 Anexo 4 - lispscript1.lisp e script1.py

lispscript1.lisp - para implementação random e genética

```
#! /usr/local/bin/clisp
; recebe como argumentos os valores dos pesos das heurísticas e o
; numero do tabuleiro a aceder.
(setf valor0 (with-input-from-string (in (nth 0 *args*))(read in)))
(setf valor1 (with-input-from-string (in (nth 1 *args*))(read in)))
(setf valor2 (with-input-from-string (in (nth 2 *args*))(read in)))
(setf valor3 (with-input-from-string (in (nth 3 *args*))(read in)))
(setf tab-pos (with-input-from-string (in (nth 4 *args*))(read in)))

; vai buscar todos os tabuleiros que estao no ficheiro "tabs_diff.txt" e guarda-os.
(setf tabuleiros (with-open-file (s "tabs_diff.txt")
 (loop with eof = '#:eof
     for object = (read s nil eof)
     until (eq object eof)
     collect object)))

(load "entregue.lisp")

; acede ao tabuleiro que esta na posicao tab-pos
(setf a1 (nth tab-pos tabuleiros))
; realiza a procura-best com 4 pecas e o tabuleiro acima referido.
(procura-best a1 '(t i l l))
```

script1.py - implementação random

```
#!/usr/bin/env python3

import random
import subprocess
import re

random.seed()

# funcao para realizar a chamada de sistema e obter o output da chamada
def system_call(command):
        p = subprocess.Popen([command], stdout=subprocess.PIPE, shell=True)
        return p.stdout.read()

while True :
        # valores de heurísticas diferentes
        heur_random_values = [random.random() for i in range(4)]

        out = open("output_random_diff.txt", "a+") # ficheiro para output de resultados
        heurísticas_string = "heuristic values: {}\n".format(heur_random_values)
        out.write(heurísticas_string)
        out.close()

        for num_tabs in range(9):
                out = open("output_random_diff.txt", "a+")
                line = open("tabs_diff.txt", "r") # ficheiro onde estao os tabuleiros
                tab = line.readlines()[num_tabs]
                line.close()

                tabuleiro_string = "tabuleiro numero: {}\n".format(num_tabs)
```

```
                out.write(tabuleiro_string)

                cmd="gtimeout 20s ./lispscript1.lisp {} {} {} {} {}".format(heur_random_values[0], heur_random_values[1],
heur_random_values[2], heur_random_values[3], num_tabs)

                cmd_result = system_call(cmd)
                print("running command...    ", cmd)
                print("result: ", cmd_result)

                result = [int(s) for s in cmd_result.split() if s.isdigit()]
                if result:
                        out.write("resultado : {}\n".format(result[0]))
                else:
                        out.write("resultado : impossivel\n")
                out.close()
```

4.6.5 Anexo 5 - Algoritmo Genético - parte 1

count	heuristica 0	heuristica 1	heuristica 2	heuristica 3	tab0	tab1	tab2	tab3	tab4	tab5	tab6	tab7	tab8	soma	Soma5
0	0,46463130	0,42268956	0,58694382	0,06780099	0	100	100	200	100	200	0	-500	0	200	1000,00
1	0,01644122	0,93107579	0,76148260	0,06780099	0	100	100	200	100	200	0	-500	0	200	
2	0,06252293	0,69841265	0,58694382	0,06780099	0	100	100	200	100	200	0	-500	0	200	
3	0,46463130	0,61696860	0,58694382	0,06311618	0	100	100	200	100	200	0	-500	0	200	
4	0,06252293	0,93107579	0,73044456	0,06780099	0	100	100	200	100	200	0	-500	0	200	
5	0,25373280	0,61696860	0,58694382	0,06311618	0	100	100	200	100	200	0	-500	0	200	1000,00
6	0,46463130	0,42268956	0,58694382	0,06311618	0	100	100	200	100	200	0	-500	0	200	
7	0,06252293	0,42268956	0,91349278	0,06311618	0	100	100	200	100	200	0	-500	0	200	
8	0,46463130	0,61696860	0,76148260	0,06251678	0	100	100	200	100	200	0	-500	0	200	
9	0,01644122	0,61696860	0,76148260	0,06311618	0	100	100	200	100	200	0	-500	0	200	
10	0,91285189	0,33836807	0,91349278	0,06780099	0	100	100	200	100	200	0	-500	0	200	1000,00
11	0,91285189	0,42268956	0,58694382	0,04369355	0	100	100	200	100	200	0	-500	0	200	
12	0,06252293	0,42268956	0,27038572	0,06311618	0	100	100	200	100	200	0	-500	0	200	
13	0,01644122	0,42268956	0,73044456	0,06251678	0	100	100	200	100	200	0	-500	0	200	
14	0,46463130	0,93107579	0,73044456	0,06780099	0	100	100	200	100	200	0	-500	0	200	
15	0,46463130	0,33836807	0,73044456	0,04369355	0	100	100	200	100	200	0	-500	0	200	1000,00
16	0,06252293	0,42268956	0,58694382	0,06311618	0	100	100	200	100	200	0	-500	0	200	
17	0,25373280	0,93107579	0,76148260	0,04369355	0	100	100	200	100	200	0	-500	0	200	
18	0,46463130	0,42268956	0,27038572	0,06251678	0	100	100	200	100	200	0	-500	0	200	
19	0,06252293	0,69841265	0,91349278	0,04369355	0	100	100	200	100	200	0	-500	0	200	
20	0,06252293	0,61696860	0,73044456	0,06251678	0	100	100	200	100	200	0	-500	0	200	1000,00
21	0,91285189	0,69841265	0,73044456	0,06251678	0	100	100	200	100	200	0	-500	0	200	
22	0,91285189	0,33836807	0,76148260	0,04369355	0	100	100	200	100	200	0	-500	0	200	
23	0,06252293	0,69841265	0,27038572	0,06780099	0	100	100	200	100	200	0	-500	0	200	
24	0,46463130	0,93107579	0,76148260	0,06251678	0	100	100	200	100	200	0	-500	0	200	
25	0,46463130	0,42268956	0,27038572	0,06780099	0	100	100	200	100	200	0	-500	0	200	1200,00
26	0,25373280	0,42268956	0,58694382	0,06311618	0	100	100	200	100	200	0	-500	0	200	
27	0,01644122	0,69841265	0,58694382	0,06311618	0	100	100	200	100	200	0	-500	0	200	
28	0,46463130	0,33836807	0,91349278	0,04369355	0	100	100	200	100	200	0	-500	0	200	
29	0,17679017	0,99800133	0,47329903	0,36889546	100	100	100	200	200	200	0	-500	0	400	
30	0,91285189	0,42268956	0,73044456	0,04369355	0	100	100	200	100	200	0	-500	0	200	1300,00
31	0,01644122	0,42268956	0,73044456	0,06780099	0	100	100	200	100	200	0	-500	0	200	
32	0,01644122	0,33836807	0,73044456	0,04369355	0	100	100	200	100	200	0	-500	0	200	
33	0,01644122	0,61696860	0,27038572	0,04369355	100	100	100	200	100	200	0	-500	0	300	
34	0,17679017	0,99800133	0,47329903	0,36889546	100	100	100	200	200	200	0	-500	0	400	
35	0,06252293	0,42268956	0,76148260	0,06251678	0	100	100	200	100	200	0	-500	0	200	1400,00
36	0,01644122	0,69841265	0,76148260	0,06251678	0	100	100	200	100	200	0	-500	0	200	
37	0,01644122	0,61696860	0,27038572	0,04369355	100	100	100	200	100	200	0	-500	0	300	
38	0,91285189	0,93107579	0,73044456	0,33585559	0	100	100	200	200	200	0	-500	0	300	
39	0,17679017	0,99800133	0,47329903	0,36889546	100	100	100	200	200	200	0	-500	0	400	
40	0,01644122	0,61696860	0,58694382	0,06251678	0	100	100	200	100	200	0	-500	0	200	1400,00
41	0,46463130	0,42268956	0,58694382	0,06311618	0	100	100	200	100	200	0	-500	0	200	
42	0,01644122	0,61696860	0,27038572	0,04369355	100	100	100	200	100	200	0	-500	0	300	
43	0,91285189	0,93107579	0,73044456	0,33585559	0	100	100	200	200	200	0	-500	0	300	
44	0,17679017	0,99800133	0,47329903	0,36889546	100	100	100	200	200	200	0	-500	0	400	

Anexo 5 - Algoritmo Genético- parte 2

45	0,01644122	0,42268956	0,27038572	0,06251678	0	100	100	200	100	200	0	-500	0	200	1400,00
46	0,46463130	0,33836807	0,27038572	0,06251678	0	100	100	200	100	200	0	-500	0	200	
47	0,01644122	0,61696860	0,27038572	0,04369355	100	100	100	200	100	200	0	-500	0	300	
48	0,91285189	0,93107579	0,73044456	0,33585559	0	100	100	200	200	200	0	-500	0	300	
49	0,17679017	0,99800133	0,47329903	0,36889546	100	100	100	200	200	200	0	-500	0	400	
50	0,25373280	0,61696860	0,76148260	0,04369355	0	100	100	200	100	200	0	-500	0	200	1400,00
51	0,01644122	0,42268956	0,76148260	0,06780099	0	100	100	200	100	200	0	-500	0	200	
52	0,01644122	0,61696860	0,27038572	0,04369355	100	100	100	200	100	200	0	-500	0	300	
53	0,91285189	0,93107579	0,73044456	0,33585559	0	100	100	200	200	200	0	-500	0	300	
54	0,17679017	0,99800133	0,47329903	0,36889546	100	100	100	200	200	200	0	-500	0	400	
55	0,46463130	0,42268956	0,91349278	0,06311618	0	100	100	200	100	200	0	-500	0	200	1400,00
56	0,91285189	0,42268956	0,58694382	0,06251678	0	100	100	200	100	200	0	-500	0	200	
57	0,01644122	0,61696860	0,27038572	0,04369355	100	100	100	200	100	200	0	-500	0	300	
58	0,91285189	0,93107579	0,73044456	0,33585559	0	100	100	200	200	200	0	-500	0	300	
59	0,17679017	0,99800133	0,47329903	0,36889546	100	100	100	200	200	200	0	-500	0	400	
60	0,46463130	0,93107579	0,73044456	0,06251678	0	100	100	200	100	200	0	-500	0	200	1500,00
61	0,01644122	0,61696860	0,27038572	0,04369355	100	100	100	200	100	200	0	-500	0	300	
62	0,91285189	0,93107579	0,73044456	0,33585559	0	100	100	200	200	200	0	-500	0	300	
63	0,91285189	0,93107579	0,73044456	0,33585559	0	100	100	200	200	200	0	-500	0	300	
64	0,17679017	0,99800133	0,47329903	0,36889546	100	100	100	200	200	200	0	-500	0	400	
65	0,06252293	0,61696860	0,91349278	0,06251678	0	100	100	200	100	200	0	-500	0	200	1500,00
66	0,01644122	0,61696860	0,27038572	0,04369355	100	100	100	200	100	200	0	-500	0	300	
67	0,91285189	0,93107579	0,73044456	0,33585559	0	100	100	200	200	200	0	-500	0	300	
68	0,91285189	0,93107579	0,73044456	0,33585559	0	100	100	200	200	200	0	-500	0	300	
69	0,17679017	0,99800133	0,47329903	0,36889546	100	100	100	200	200	200	0	-500	0	400	
70	0,01644122	0,93107579	0,91349278	0,06251678	0	100	100	200	100	200	0	-500	0	200	1500,00
71	0,01644122	0,61696860	0,27038572	0,04369355	100	100	100	200	100	200	0	-500	0	300	
72	0,91285189	0,93107579	0,73044456	0,33585559	0	100	100	200	200	200	0	-500	0	300	
73	0,91285189	0,93107579	0,73044456	0,33585559	0	100	100	200	200	200	0	-500	0	300	
74	0,17679017	0,99800133	0,47329903	0,36889546	100	100	100	200	200	200	0	-500	0	400	
75	0,01644122	0,61696860	0,27038572	0,04369355	100	100	100	200	100	200	0	-500	0	300	1600,00
76	0,91285189	0,93107579	0,73044456	0,33585559	0	100	100	200	200	200	0	-500	0	300	
77	0,91285189	0,93107579	0,73044456	0,33585559	0	100	100	200	200	200	0	-500	0	300	
78	0,01644122	0,69841265	0,27038572	0,04369355	100	100	100	200	100	200	0	-500	0	300	
79	0,17679017	0,99800133	0,47329903	0,36889546	100	100	100	200	200	200	0	-500	0	400	
80	0,91285189	0,93107579	0,73044456	0,33585559	0	100	100	200	200	200	0	-500	0	300	1600,00
81	0,91285189	0,93107579	0,73044456	0,33585559	0	100	100	200	200	200	0	-500	0	300	
82	0,01644122	0,69841265	0,27038572	0,04369355	100	100	100	200	100	200	0	-500	0	300	
83	0,98608646	0,78588903	0,26342241	0,27516649	0	100	100	200	200	200	0	-500	0	300	
84	0,17679017	0,99800133	0,47329903	0,36889546	100	100	100	200	200	200	0	-500	0	400	
85	0,98608646	0,78588903	0,26342241	0,27516649	0	100	100	200	200	200	0	-500	0	300	1700,00
86	0,01644122	0,61696860	0,27038572	0,04369355	100	100	100	200	100	200	0	-500	0	300	
87	0,02157429	0,26783684	0,00426163	0,07152764	100	100	100	200	100	200	0	-500	0	300	
88	0,17679017	0,99800133	0,47329903	0,36889546	100	100	100	200	200	200	0	-500	0	400	
89	0,25373280	0,93107579	0,27038572	0,33585559	100	100	100	200	200	200	0	-500	0	400	

4.6.6 Anexo 6 - lispscript2.lisp e script2.py

script2.py - implementação do Algoritmo genético

```
#!/usr/bin/env python3

import random
import subprocess

random.seed()

def system_call(command):
    p = subprocess.Popen([command], stdout=subprocess.PIPE, shell=True)
    return p.stdout.read()

class TesteHeuristica:
    def __init__(self):
        self.soma = 0
        self.heurísticas = [random.random() for i in range(4)]
        self.tabuleiros = []

    def set_heurísticas(self, valor0, valor1, valor2, valor3):
        self.heurísticas = [valor0, valor1, valor2, valor3]


# MELHORES VALORES DO RANDOM
melhor0 = TesteHeuristica()
melhor0.tabuleiros = [0.00 , 100.00 , 100.00, 200.00, 100.00 , 200.00, 0.00 , -500.00, 0.00]
melhor0.soma = sum(melhor0.tabuleiros)
print(melhor0.soma)
melhor0.set_heurísticas(0.06252293039487711, 0.6169685981711657 , 0.5869438219752874 , 0.04369354680040316 )

melhor1 = TesteHeuristica()
melhor1.tabuleiros = [0.00,100.00,100.00,200.00,100.00,200.00,0.00,-500.00,0.00]
melhor1.soma = sum(melhor1.tabuleiros)
print(melhor1.soma)
melhor1.set_heurísticas(0.9128518893940584 , 0.4226895567776138 , 0.9134927770474299 , 0.06251677931489352 )

melhor2 = TesteHeuristica()
melhor2.tabuleiros = [0.00,100.00,100.00,200.00,100.00,200.00,0.00,-500.00,0.00]
melhor2.soma = sum(melhor2.tabuleiros)
print(melhor2.soma)
melhor2.set_heurísticas(0.25373280159762823 , 0.9310757885884794 , 0.7614826005183634 , 0.06311617522444746 )

melhor3 = TesteHeuristica()
melhor3.tabuleiros = [0.00,100.00,100.00,200.00,100.00,200.00,0.00,-500.00,0.00]
melhor3.soma = sum(melhor3.tabuleiros)
print(melhor3.soma)
melhor3.set_heurísticas(0.016441216301480432 , 0.3383680738360155 , 0.2703857247354139 , 0.06780098767615228 )

melhor4 = TesteHeuristica()
melhor4.tabuleiros = [-500.00,100.00,100.00,200.00,200.00,200.00,0.00,-500.00,0.00]
melhor4.soma = sum(melhor4.tabuleiros)
print(melhor4.soma)
melhor4.set_heurísticas(0.4646313036167071 , 0.6984126477276682 , 0.7304445618597235 , 0.3358555934538632 )

melhores = [melhor0, melhor1, melhor2, melhor3, melhor4]
```

```
class Genetico:
	def __init__(self, melhores):
		self.melhores = melhores
		self.totais = melhores

	def crossover(self):
		"retorna um filho dos melhores"
		heurística0 = [teste.heurísticas[0] for teste in self.melhores]
		heurística1 = [teste.heurísticas[1] for teste in self.melhores]
		heurística2 = [teste.heurísticas[2] for teste in self.melhores]
		heurística3 = [teste.heurísticas[3] for teste in self.melhores]

		filho = TesteHeuristica()
		filho.set_heurísticas(heurística0[random.randint(0,4)], heurística1[random.randint(0,4)],
heurística2[random.randint(0,4)], heurística3[random.randint(0,4)])
		return filho

iteracoes_genetico = 0

while True :
	# para valores de heurísticas diferentes
	iteracoes_genetico += 1
	print("iteracao: ", iteracoes_genetico)

	out = open("output_random_diff.txt", "a+")
	heurísticas_string = "\niteracao: {}\n".format(iteracoes_genetico)
	#out.write(heurísticas_string)
	out.close()

	genetico = Genetico(melhores)

	gerados = [genetico.crossover() for i in range(9)] # cria 9 filhos a partir dos melhores

	random_filho = TesteHeuristica() # cria um elemento random

	# calcula a soma dos valores nos 9 tabuleiros para uma dada heurística
	# penalizando o facto de nao acabar em -500

	for filho in gerados:
		heur_random_values = filho.heurísticas

		out = open("output_random_diff.txt", "a+")
		heurísticas_string = "heuristic values: {}\n".format(heur_random_values)
		#out.write(heurísticas_string)
		out.close()

		for num_tabs in range(9):
			out = open("output_random_diff.txt", "a+")
			line = open("tabs_diff.txt", "r")
			tab = line.readlines()[num_tabs]
			# out.write("{}".format(tab))
			line.close()

			tabuleiro_string = "tabuleiro numero: {}\n".format(num_tabs)

			cmd="gtimeout 20s ./lispscript1.lisp {} {} {} {} {}".format(heur_random_values[0], heur_random_values[1],
heur_random_values[2], heur_random_values[3], num_tabs)
```

```
                        cmd_result = system_call(cmd)

                        result = [int(s) for s in cmd_result.split() if s.isdigit()]
                        if result:
                                filho.tabuleiros += [result[0]]
                                filho.soma += result[0]
                        else:
                                filho.tabuleiros += [-500]
                                filho.soma -= 500
                        out.close()

                filho_string = "filho: {} , soma: {} , heurísticas: {}\n".format(filho.tabuleiros,filho.soma, filho.heurísticas)
                print(filho_string)
                genetico.totais += [filho]

        # RANDOM
        heur_random_values = random_filho.heurísticas

        out = open("output_random_diff.txt", "a+")
        heurísticas_string = "heuristic values: {}\n".format(heur_random_values)
        out.close()

        for num_tabs in range(9):
                out = open("output_random_diff.txt", "a+")
                line = open("tabs_diff.txt", "r")
                tab = line.readlines()[num_tabs]
                line.close()

                tabuleiro_string = "tabuleiro numero: {}\n".format(num_tabs)

                cmd="gtimeout 20s ./lispscript1.lisp {} {} {} {} {}".format(heur_random_values[0], heur_random_values[1],
heur_random_values[2], heur_random_values[3], num_tabs)

                cmd_result = system_call(cmd)

                result = [int(s) for s in cmd_result.split() if s.isdigit()]
                if result:
                        random_filho.tabuleiros += [result[0]]
                        random_filho.soma += result[0]
                        print(result[0])
                else:
                        random_filho.tabuleiros += [-500]
                        random_filho.soma -= 500
                        print(-500)
                out.close()

        random_filho_string = "filho: {} , soma: {} , heurísticas: {}\n".format(random_filho.tabuleiros, random_filho.soma,
random_filho.heurísticas)
        print(random_filho_string)

        genetico.totais += [random_filho]

        genetico.totais = sorted(genetico.totais, key=lambda elem: elem.soma)

        print("melhores 5:")
        genetico.melhores = genetico.totais[-5:]
        out = open("output_random_diff.txt", "a+")
```

```
for elem in genetico.melhores:
        print("tabuleiros:",elem.tabuleiros, "\n", "soma:",elem.soma, "\n", "heurísticas:", elem.heurísticas, "\n")
        out.write("filho: {} , soma: {} , heurísticas: {}\n".format(elem.tabuleiros, elem.soma, elem.heurísticas))
out.close()
```