Grupo Nº 34

Inteligência Artificial

1.º Semestre 2015/2016

IA-Tetris

Relatório de Projecto

João Pedro Martins Serras nº 79664

Daniel José Matias Caramujo nº 79714

Francisco Miguel Polaco Santos nº79719

Índice

1 Implementação Tipo Tabuleiro e Funções do problema de Procura

1.1 Tipo Abstracto de Informação Tabuleiro

Para a representação do tipo tabuleiro foi escolhida uma lista de listas. Esta lista é constituída por dezoito listas representando as linhas, em que estas têm um tamanho de dez para representar as diversas colunas. Cada elemento destas dezoito listas pode tomar o valor de *nil* ou *t,* simulando a posição estar desocupada ou ocupada, respetivamente. Esta representação é conceptualmente parecida com um *array* de duas dimensões, no entanto muito diferente a nível computacional.

A escolha desta implementação deveu-se muito ao facto de que a linguagem *ANSI Common Lisp* ter muitas funções sobre listas e toda a linguagem trabalhar à sua volta o que a torna optimizada para esta estrutura. Embora a representação por *array* bidimensional pudesse ser a mais eficiente nas operações, a complexidade de código para gestão desta estrutura seria superior às listas. Assim, conseguimos implementar o tipo tabuleiro com pouco código e alto desempenho.

1.2 Implementação de funções do problema de procura

A implementação da função *accoes* recorremos à análise de todas as rotações possíveis para a próxima peça a ser colocada. Para isso, verifica-se qual é o primeiro elemento da lista de peças por colocar e com uma estrutura de *if then else* obtém-se uma lista de rotações possíveis. Aqui poderíamos ter optado por criar uma *hash table* que ao símbolo estivesse associado a lista, no entanto esta tabela teria de ser instanciada na função o que faria perder tempo com a construção dela. A mesma tabela instanciada globalmente ao programa não valeria de nada, visto o ambiente de testes encerrar o programa cada vez que testa algo novo. Assim, tendo a lista de rotações para cada coluna do tabuleiro, até à coluna que somada com a largura da peça iguala a última coluna do tabuleiro (inclusive) são geradas todas as acções, listadas e retornadas.

A função *resultado* só por si tem algum grau de dificuldade de implementação. Ao longo do desenvolvimento do projecto houve a necessidade de corrigir algumas vezes esta função devido aos pormenores inerentes à colocação das peças num tabuleiro de *Tetris*. A ideia da

função é percorrer as colunas entre o início da peça e o seu fim e descobrir qual a altura máxima do tabuleiro naquela zona. Com esta informação “obriga-se” a base da peça a ser colocada na altura dada pela expressão $h_{coluna} - h_{peça}$ (o mesmo seria dizer que estamos a colocar o topo

da peça sobre a coluna mais alta). Seguidamente testa-se se a peça é possível de inserir (verifica-se se não ficam sobrepostas peças do tabuleiro com a nova), se for introduz-se e quebra-se o ciclo, caso contrário incrementa-se a altura a ser colocada e tenta-se novamente. Posteriormente, se o topo do tabuleiro estiver preenchido é retornado apenas o novo estado, caso contrário calcula-se também os pontos. Este cálculo soma aos pontos do estado anterior os pontos dados pelo número de linhas removidas.

Este método foi escolhido por ser mais eficiente. Em vez de ter que descer a peça desde o topo do tabuleiro até esta colidir, tentamos colocar imediatamente a peça no sítio (com sucesso em casos ótimos. Em caso de insucesso, o pior caso é repetir a altura da peça, ao contrário do método convencional, no qual no pior caso seria preciso repetir a altura do tabuleiro).

Vejamos um exemplo do caso óptimo e do pior caso:

Caso óptimo (coloca à primeira tentativa):

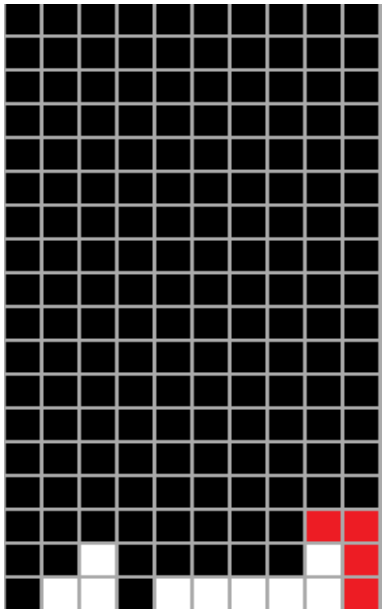

Figura 1.2-A

Pior caso (repete a altura da peça):

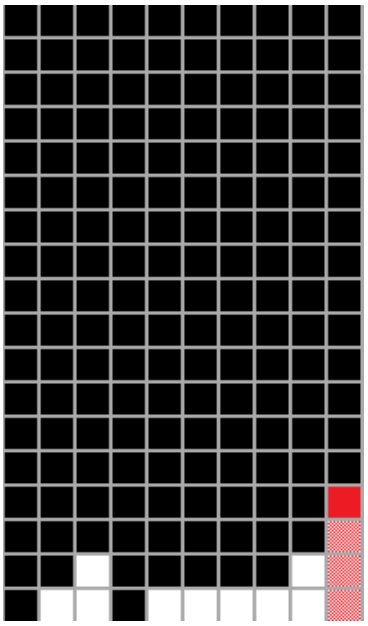

Figura 1.2-B

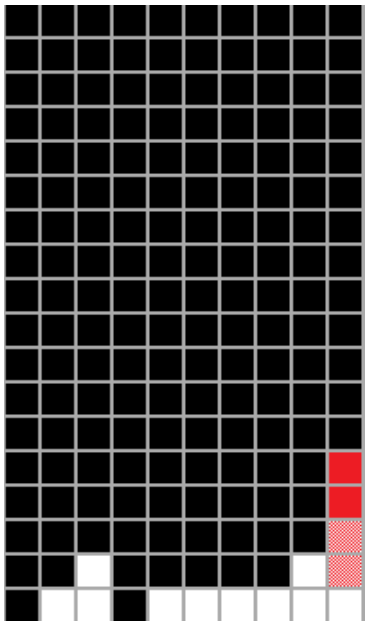

Figura 1.2-C

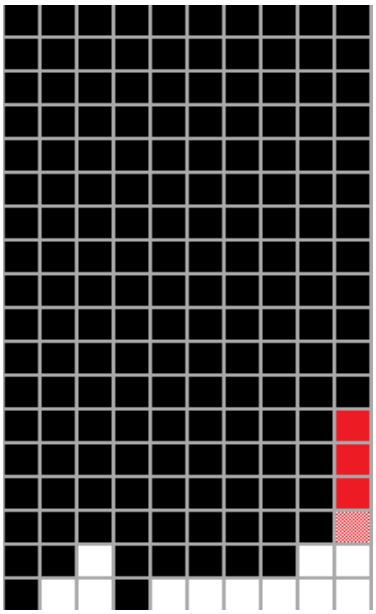

Figura 1.2-D

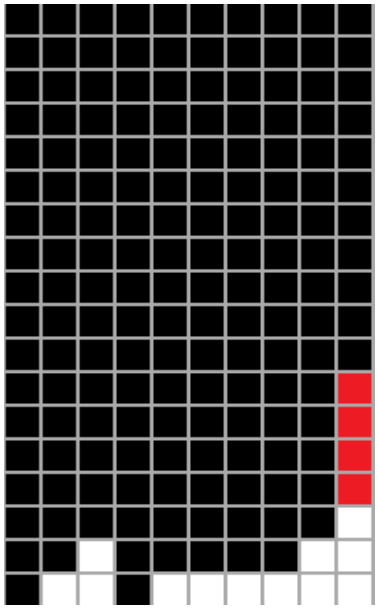

Figura 1.2-E

No entanto, esta implementação tem um caso particular bastante problemático, que não é testado nos testes da plataforma *Mooshak*. Isto dá-se quando ao colocar uma peça (não acontece com todas, apenas um numero limitado de peças com uma determinada rotação) esta encaixa, mas não devia. Vejamos as seguintes imagens para facilitar a compreensão.

Caso problemático:

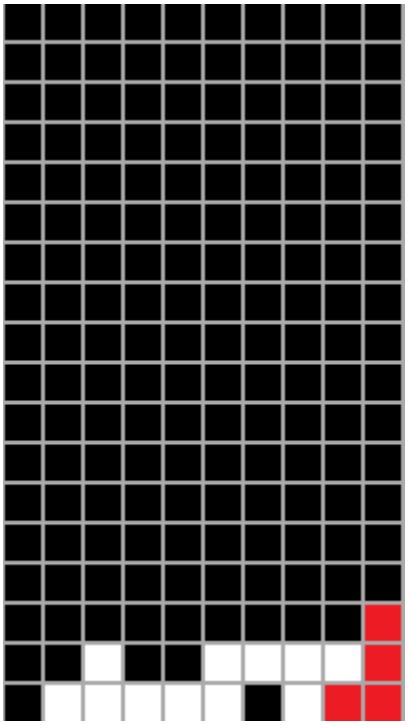

Figura 1.2-F

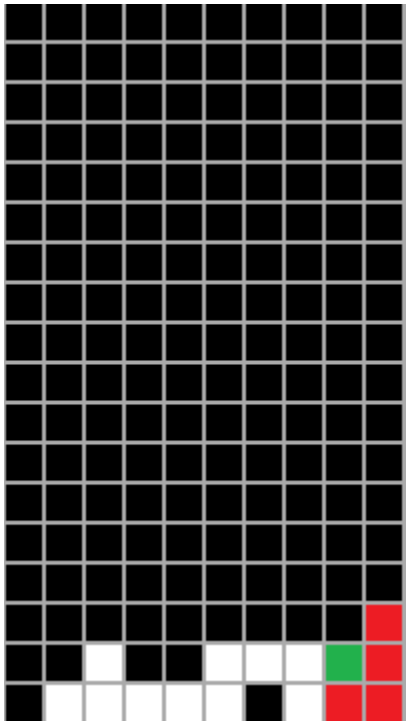

Figura 1.2-G

Como podemos ver, seguindo o método descrito acima, a peça encaixa à primeira tentativa, no entanto não devia, dado que não conseguiria passar, pelo método tradicional (quadrado a verde). Este problema foi resolvido vendo se a peça colocada é uma das que pode causar conflito, e verificando se não existem zonas preenchidas quando não deviam.

2 Implementação Algoritmos de Procura

2.1 Procura-pp

O algoritmo da procura em profundidade primeiro foi implementado de uma forma recursiva. Como condição de paragem da recursão é verificado se o estado é solução. Se a condição se verificar é retornado para o nível de recursão anterior uma lista com a acção efetuada para chegar ao estado. Caso contrário é feita a geração dos nós filhos usando a lista de acções possíveis e chamada a função sobre cada filho gerado, um de cada vez afundando assim na árvore. À medida que a recursão é resolvida, a lista de acções é criada e por fim retornada. Esta parte do programa é constituída por duas funções, uma que apenas serve para iniciar a recursão (*procura-pp*) e outra (função interna lambda) que realmente executa o algoritmo em si (*procura-pp-aux*).

2.2 Procura-A*

Semelhante à procura de cima, esta secção é constituída por duas funções, no entanto foram adicionados tipos abstratos de informação (TAI) com o objetivo de facilitar a escrita deste algoritmo. O TAI *no* representa um nó na árvore de pesquisa, internamente e usado um par com o estado e o seu valor da função *f.* São utilizadas duas estruturas para encapsular outros conceitos. A estrutura *capsula* representa o retorno da função *procura-A*-aux,* visto esta necessitar de retornar dois valores. A segunda estrutura denominada *sucessor* serve representar o nó gerado e a acção para lá chegar.

O algoritmo escolhido para implementar a pesquisa *A** foi a *RBFS* devido à sua natureza recursiva que permite que o espaço de memória permaneça linear durante toda a procura. Assim, para começar é efetuado um teste do caso de paragem de recursão, que se tiver sucesso retorna a acção efetuada para chegar ao estado e o valor da função *f* usado para lá chegar. Caso contrário, serão gerados os sucessores tendo em conta o valor da função *f*. Se não for possível gerar sucessores, a função retorna falha.

Posteriormente, começa um *loop* em que se escolhe sempre o melhor sucessor da lista e se expande tendo em conta o valor de *f* do segundo melhor sucessor. Para encontrar estes valores, não é utilizada uma ordenação, visto o critério de desempate implicar que seja sempre

necessário chegar ao último elemento da lista. Assim sendo, procurar o último valor mais baixo torna o problema, no pior caso $O(n + \log_2(n))$. Encontrando apenas o valor pretendido a complexidade é, no pior caso $O(n)$.

A função é recursivamente chamada sobre o sucessor escolhido e espera-se o resultado, se este for bem-sucedido então constrói-se a lista de acções. Esta lista é também construída à medida que os níveis de recursão são resolvidos.

3 Funções Heurísticas

3.1 Heurística 1 - Aplana

3.1.1 Motivação

A ideia desta heurística consiste em, como o nome sugere, aplanar o mais possível o tabuleiro de jogo, ou seja, fazer com que a altura das colunas seja o menor possível. Esta ideia surge por dois motivos, sendo o primeiro o facto de que ao aplanar, há linhas que são removidas, significando isto que são ganhos pontos, e o segundo o facto de isto originar mais espaço no tabuleiro, facilitando a colocação de peças e diminuindo o risco de perder por passar da linha limite.

Para calcular esta heurística nada mais é preciso para além da informação do tabuleiro. Esta é precisa para aceder à informação da altura de cada coluna.

Dada a simplicidade desta heurística não houve necessidade de ensaiar variantes.

3.1.2 Forma de Cálculo

O cálculo desta heurística é bastante simples. Consiste apenas em calcular o somatório da altura das colunas, com vista a obter um valor a ser minimizado nas funções de procura. Traduzindo numa fórmula matemática aplicável:

$$\sum_{i=1}^{nº\ colunas} (Altura_i)$$

Para facilitar a compreensão, consideremos o seguinte exemplo:

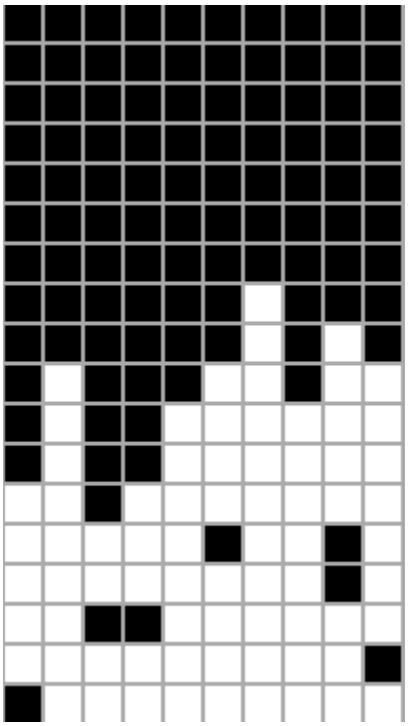

Figura 3.1-A

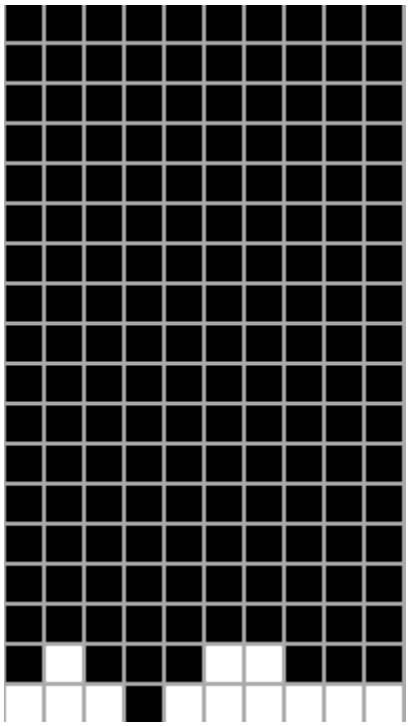

Figura 3.1-B

Como podemos ver, a situação ilustrada na figura 3.1-A não é ideal, e dificulta a obtenção de bons resultados, ao contrário da situação em 3.1-B. Aplicando a heurística:

Para 3.1-A:

$$\sum_{i=1}^{10}(Altura_i) = 6 + 9 + 5 + 6 + 8 + 9 + 11 + 8 + 10 + 9 = 81$$

Para 3.1-B:

$$\sum_{i=1}^{10}(Altura_i) = 1 + 2 + 1 + 0 + 1 + 2 + 2 + 1 + 1 + 1 = 12$$

Podemos constatar que esta heurística devolve a soma de todas as colunas, sendo que um menor valor traduz-se numa situação mais favorável.

3.2 Heurística 2 – Uniformiza

3.2.1 Motivação

Esta heurística tem por base a ideia de que é mais fácil colocar peças num tabuleiro liso do que num tabuleiro rugoso, ou seja, num tabuleiro em que a discrepância de uma coluna para a outra não é grande. Se essa discrepância for grande dá-se a criação de "poços", que são sítios onde as peças não conseguem entrar (excepto a peça "i" em pé, mas mesmo esta pode não ter qualquer vantagem em entrar num poço se houverem outros que impossibilitem a obtenção de pontos). Assim sendo, quanto mais liso for o tabuleiro, maior a facilidade de colocar a peça, independentemente desta.

Para o cálculo desta heurística é apenas preciso o acesso à informação do tabuleiro, para de lá obter a altura das colunas.

Esta heurística apresenta um desafio de implementação, no cálculo individual da discrepância, que influencia o resultado final. Para solucionar este problema foi preciso ter em conta o sinal dos valores, (positivo ou negativo). Essa solução é explicada mais detalhadamente na secção seguinte, forma de cálculo.

3.2.2 Forma de Cálculo

O cálculo desta heurística é relativamente simples. Consiste em calcular o somatório do módulo da subtração de uma coluna com a sua subsequente. A subtração dá-nos a discrepância entre duas colunas. O módulo desta assegura-nos que o valor é positivo, independentemente de ser a primeira maior que a segunda ou o contrário, isto porque caso as colunas estivessem em "escada ascendente" o valor final iria dar negativo, e visto que a função será minimizada na procura, o resultado não seria o desejável. Traduzindo numa fórmula matemática:

$$\sum_{i=1}^{nº\ colunas-1}(|Altura_i - Altura_{i+1}|)$$

Para facilitar a compreensão, consideremos o seguinte exemplo:

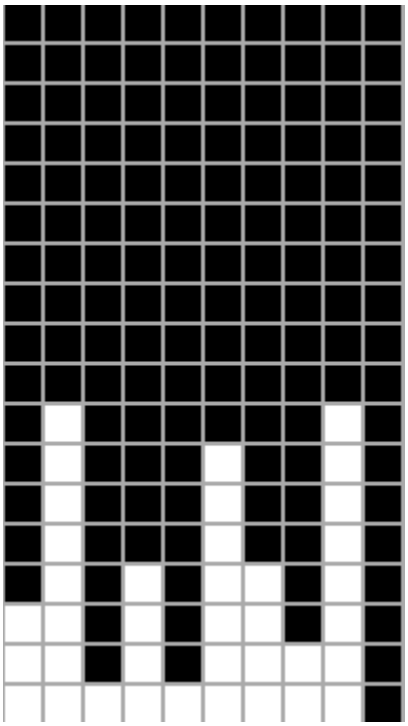

Figura 3.2-A

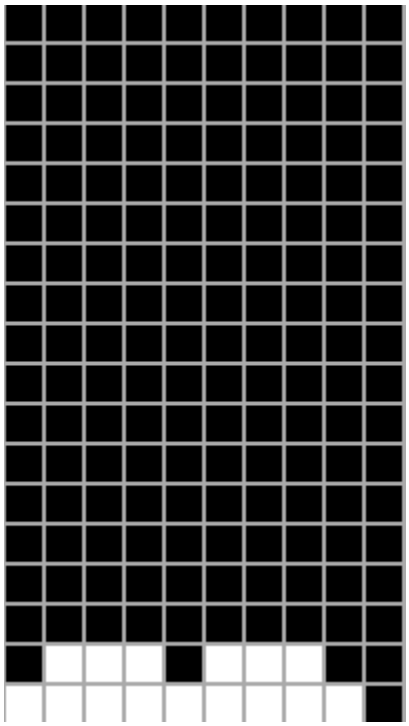

Figura 3.2-B

Como podemos observar, o caso ilustrado em 3.2-A dificulta a colocação de peças, ao contrário do 3.2-B. Aplicando a heurística:

Para 3.2-A

$$\sum_{i=1}^{9}(|Altura_i - Altura_{i+1}|) = |3-8| + |8-1| + |1-4| + |4-1| + |1-7| + |7-4| + |4-2| + |2-8| + |8-0| = 43$$

Para 3.2-B

$$\sum_{i=1}^{9}(Altura_i - Altura_{i+1}|\,) = |1-2| + |2-2| + |2-2| + |2-1| + |1-2| + |2-2| + |2-2| + |2-1| + |1-0| = 5$$

Podemos observar que esta heurística traduz uma situação favorável num número de baixo valor.

3.3 Heurística 3 – Best

3.3.1 Motivação

Esta heurística é simplesmente a junção de várias heurísticas numa só, com coeficientes que estabelecem prioridades entre elas de forma a maximizar o desempenho.

Para a aplicar precisamos do estado completo, mas apenas para passar na chamada de outras funções heurísticas, não sendo feito nenhum acesso a este dentro desta função.

Esta heurística manteve sempre a mesma estrutura. As variâncias observadas foram com o teste de várias funções heurísticas, que ultimamente não chegaram à versão final, e os correspondentes coeficientes.

3.3.2 Forma de Cálculo

O cálculo desta heurística consiste apenas da soma dos valores retornados das funções heurísticas, multiplicadas pelo coeficiente de prioridade. Traduzindo para uma fórmula matemática:

$$\sum_{i=1}^{nº\ heurísticas} (Coeficiente_i \text{ x } Heurística_i)$$

No caso concreto que foi desenvolvido, isto traduz-se na soma das duas heurísticas vistas em 3.1 com 3.2, multiplicando cada uma pelo respetivo coeficiente. A obtenção dos coeficientes foi conseguida através de um estudo empírico da variação do resultado com diferentes combinações de coeficientes, como está ilustrado na figura 3.3-A.

CoefH1	CoefH2	Tempo Decorrido	Pontuacao
1	0	21,524	600
0	1	20,012	600
1	1	22,59	600
2	1	24,704	600
3	1	27,92	600
4	1	27,25	600
1	2	26,52	600
1	3	26,62	600
1	4	25,61	600
10	15	1,64	600
15	10	0,7382	600
20	10	0,3	600
30	10	0,236	500
40	10	0,209	500
50	10	0,08	500
10	30	1,08	600

Figura 3.3-A Custo-Qualidade

Com este estudo concluímos que, para este caso específico, os melhores coeficientes são 20 e 10, para *H1 (Aplana)* e *H2 (Uniformiza)*, respectivamente. Assim sendo, podemos traduzir no seguinte:

$$\sum_{i=1}^{2}(Coeficiente_i \text{ x } Heurística_i) = 20\ x\ Aplana(estado) + 10 \text{ x } Uniformiza(estado)$$

Isto garante o melhor balanço na utilização das duas heurísticas em simultâneo.

4 Estudo Comparativo

4.1 Estudo Algoritmos de Procura

4.1.1 Critérios a analisar

Os critérios usados para comparar os algoritmos de procura foram o tempo de execução, os pontos obtidos e a memória utilizada durante a execução, sendo dada especial importância aos dois primeiros.

A escolha destes critérios prende-se precisamente pelo objetivo do programa desenvolvido, que é resolver o jogo do *Tétris* com o maior número de pontos possível em tempo útil e limitado. Assim sendo é crucial que o algoritmo de procura tenha ambos estes aspetos em conta.

Também importante é o uso de memória, uma vez que esta é finita, limitada pelo sistema de submissões *Mooshak* e no mundo real os computadores têm uma memória limitada. Assim sendo é importante que o uso de memória por parte do algoritmo não comprometa o programa.

4.1.2 Testes Efectuados

Para testar os algoritmos implementados foi feito um teste, baseado no teste 25 do *Mooshak*, no qual foram medidos os dados relativos aos critérios definidos. O teste consiste em colocar num tabuleiro, já preenchido em alguns sítios, uma dada lista de peças. A lista de peças aumenta, com uma nova peça, para cada iteração do teste, sendo o tabuleiro inicial reposto. No caso do algoritmo de procura *A**, foi usada uma heurística constante de valor 0.

Este teste permite-nos aumentar a complexidade da jogada rapidamente, pondo à prova os três critérios.

4.1.3 Resultados Obtidos

	Baseado no Teste 25 do Mooshack											
	Tempo em segundos		Pontos				Memória					
			Pontos Adquiridos		Pontos Desperdiçados		Bytes		kBytes		Easy Reading	
	A*	pp	A*	pp	A*	pp	A*	pp	A*	pp	A*	pp
	Segundos	Segundos	Pontos	Pontos	Desperdiçados	Desperdiçados	Bytes	Bytes	kBytes	kBytes		kB
(t)	0,04	0	100	0	200	300	242248	17264	236.5703125	16.859375	236.57kB	16.86
(t i)	0,06	0	300	0	800	1100	4275456	27224	4175.25	26.5859375	4.08MB	26.59
(t i l)	0,116	0	600	0	1000	1600	9479304	44576	9257.132813	43.53125	9.04MB	43.53
(t i l l)	2,932	0	600	0	1500	2100	461913649	61960	451087.5391	60.5078125	440.52MB	60.51
(t i l l s)	57,272	0	600	0	1800	2400	8801341816	71440	8595060.367	69.765625	8.20GB	70
(t i l l s o)	109,228	0	700	0	2000	2700	16874230608	78952	16478740.83	77.1015625	15.71GB	77.1

Figura 4.1-A

4.1.4 Comparação dos Resultados Obtidos

Os dois tipos de dados que se destacam neste teste são o tempo e os pontos adquiridos. Isto porque, em ambas, a procura em *profundidade primeiro (pp)* mantém-se constantemente a 0. Isto deve-se ao facto de esta procura cega estar apenas interessada em colocar a peça no primeiro sítio que der. Assim, apesar de um bom desempenho temporal, tem um péssimo desempenho em termos de obtenção de pontos. O mesmo não se verifica com a *A**, que apesar de ter piores resultados temporais, tem um excelente desempenho na obtenção de pontos, uma vez que esta procura tem em conta esse fator, fazendo uma maximização dos pontos possíveis de ganhar.

Quanto ao uso de memória, a procura em profundidade primeiro mantém-se com muito pouco gasto, sempre na ordem dos *KB*, enquanto que a *A** escala neste aspeto rapidamente, atingindo a ordem dos *GB*. Isto é um problema considerável no desempenho da procura *A**. (Estes dados consistem no somatório de todo o espaço alocado ao longo do decorrer do algoritmo, sendo contabilizado independentemente de ter sido libertado posteriormente, ou de ter sido alocado repetidamente).

No entanto, comparando estas duas procuras, e pesando a importância dos critérios em análise, concluímos que a procura *A** é mais desejável do que a procura em profundidade primeiro, uma vez que esta nos garante a obtenção de pontos, apesar de um pior desempenho

temporal e espacial. Um bom desempenho, no entanto, não vale de nada se não conseguir entregar os resultados pretendidos, sendo exatamente essa a situação aqui presente.

4.2 Estudo funções de custo/heurísticas

4.2.1 Critérios a analisar

Para a avaliação tanto das funções de custo, como das funções heurísticas, foram estabelecidos critérios de tempo e pontuação, sendo o objetivo minimizar o tempo e maximizar a pontuação. Isto porque são estes fatores que realmente importam nesta fase, de forma a melhorar o desempenho do programa cumprindo o objetivo.

4.2.2 Testes Efectuados

Para o estudo das funções de custo e das funções heurísticas foi usado o teste 25 do *Mooshak*. Achamos que este teste impõe a exigência necessária para tirarmos conclusões do seu resultado, uma vez que este teste exige um certo engenho por parte do algoritmo, na colocação das peças, para obter um bom resultado na pontuação.

O teste consiste em colocar uma dada lista de peças num tabuleiro previamente preenchido em determinadas posições. Este foi corrido para os diferentes valores dos coeficientes das heurísticas (permitindo testá-las tanto individualmente como em conjunto), usando a função *custo-oportunidad*e e novamente usando a função qualidade.

Assim conseguimos testar todas as combinações possíveis entre funções heurísticas e funções de custo de forma a determinar qual o valor ideal dos coeficientes heurísticos e qual a função de custo com melhor desempenho.

4.2.3 Resultados Obtidos

CoefH1	CoefH2	Tempo Decorrido	Pontuacao
1	0	21,524	600
0	1	20,012	600
1	1	22,59	600
2	1	24,704	600
3	1	27,92	600
4	1	27,25	600
1	2	26,52	600
1	3	26,62	600
1	4	25,61	600
10	15	1,64	600
15	10	0,7382	600
20	10	0,3	600
30	10	0,236	500
40	10	0,209	500
50	10	0,08	500
10	30	1,08	600

Figura 4.2-A Custo-Qualidade

CoefH1	CoefH2	Tempo Decorrido	Pontuacao
1	0	0,044	400
0	1	0,04	400
1	1	0,044	400
2	1	0,044	400
3	1	0,044	400
4	1	0,044	400
1	2	0,036	400
1	3	0,044	400
1	4	0,044	400
10	15	0,044	400
15	10	0,044	400
20	10	0,044	300
30	10	0,04	300
40	10	0,044	300
50	10	0,04	300
10	30	0,044	300

Figura 4.2-B Qualidade

4.2.4 Comparação dos Resultados Obtidos

Começando por analisar os resultados obtidos usando a função *custo-qualidade* (figura 4.2-A) podemos observar que as heurísticas *H1* e *H2* (vistas anteriormente no ponto 3) sozinhas não têm um bom desempenho (multiplicadas pelos coeficientes 0 e 1, alternadamente)[1]. Mantendo um das heurísticas com coeficiente constante e dando valores à outra apenas piora o desempenho do programa a nível temporal, não tendo qualquer efeito no resultado obtido nos pontos (podemos observar isto para ambas as heurísticas). Foi preciso um compromisso entre os dois coeficientes de forma a chegar ao pretendido.

Assim sendo, empiricamente chegámos ao valor 20 para o coeficiente de *H1* e 10 para o de *H2*. Este garante uma boa marca temporal sem comprometer o resultado dos pontos obtidos, visto que a *H1* tem por objetivo baixar a altura do tabuleiro, implicando que sejam feitos pontos ao eliminar as linhas.

[1] Na realidade, para os testes foi comentado a parte do código que chamava a função e multiplicava o valor pelo coeficiente, para não gastar tempo a calcular.

Olhando agora para a variação dos coeficientes heurísticos usando a função qualidade, podemos constatar que a influência destes no seu desempenho é redundante. O desempenho temporal é superior devido à simplicidade do cálculo desta função, que consiste apenas numa multiplicação enquanto que na função de *custo-oportunidade* é percorrida uma lista. O desempenho em termos de pontuação é baixo porque esta função tem por objetivo maximizar os pontos numa única jogada, enquanto a função *custo-oportunidade* tenta minimizar o desperdício de pontos globalmente, conseguindo assim num conjunto de jogadas obter uma pontuação maior do que a função qualidade no total das jogadas individuais.

Apesar de esta função de custo ter um melhor desempenho temporal, não maximiza a obtenção de pontos, deixando a função *custo-oportunidade* mais favorável à aplicação do programa.

4.3 Escolha da procura-best

De forma a criar o melhor jogador optámos por criar a função *procura-best* consistindo da procura *A** guiada pela função de custo-oportunidade, com a heurística *best* com coeficientes heurísticos 20 para *H1 (heurística-aplana)* e 10 para *H2 (heurística-uniformiza)*.

Este método garante um bom desempenho do agente, como vimos anteriormente nos pontos 4.1 e 4.2.

Vimos que a pesquisa *A** se torna bastante poderosa com auxílio das funções associadas, e sem dificuldade supera pesquisas cegas, sendo neste contexto o melhor algoritmo de procura a usar.

Vimos também que a função custo-oportunidade garante um bom desempenho na obtenção de pontos, visto o seu objetivo de, com o auxílio da pesquisa A*, minimizar o desperdicio de pontos, e ainda que a heurística *best*, ao proporcionar uma boa organização do tabuleiro de jogo, garante um bom desempenho temporal.

Concluímos então que a melhor opção seria juntar estes elementos e obtivemos confirmação disso ao passar com sucesso a todos os testes propostos.