Grupo Nº 23



**Inteligência Artificial**

1.º Semestre 2015/2016

**IA-Tetris**

Relatório de Projecto

Índice

[1 Implementação Tipo Tabuleiro e Funções do problema de Procura 4](#_Toc437374271)

[1.1 Tipo Abstracto de Informação Tabuleiro 4](#_Toc437374272)

[1.2 Implementação de funções do problema de procura 4](#_Toc437374273)

[1.2.1 Função *solucao* 4](#_Toc437374274)

[1.2.2 Função *accoes* 4](#_Toc437374275)

[1.2.3 Função *resultado* 5](#_Toc437374276)

[1.2.4 Função *coloca-peca!* 5](#_Toc437374277)

[2 Implementação Algoritmos de Procura 5](#_Toc437374278)

[2.1 Procura-pp 6](#_Toc437374279)

[2.2 Procura-A\* 6](#_Toc437374280)

[2.3 Têmpera simulada 7](#_Toc437374281)

[3 Funções Heurísticas 7](#_Toc437374282)

[3.1 Espaços-livres-totais 8](#_Toc437374283)

[3.1.1 Motivação 8](#_Toc437374284)

[3.1.2 Forma de Cálculo 8](#_Toc437374285)

[3.2 – Linhas-vazias 9](#_Toc437374286)

[3.2.1 Motivação 9](#_Toc437374287)

[3.2.2 Forma de Cálculo 9](#_Toc437374288)

[3.3 Buracos totais 10](#_Toc437374289)

[3.3.1 Motivação 10](#_Toc437374290)

[3.3.2 Forma de Cálculo 11](#_Toc437374291)

[3.4 Media-altura-colunas, altura-colunas-regiões 12](#_Toc437374292)

[3.4.1 Motivação 12](#_Toc437374293)

[3.4.2 Forma de Cálculo 12](#_Toc437374294)

[3.5 Bloco-mais-alto 13](#_Toc437374295)

[3.5.1 Motivação 13](#_Toc437374296)

[3.5.2 Forma de Cálculo 14](#_Toc437374297)

[3.6 Irregularidades 14](#_Toc437374298)

[3.6.1 Motivação 14](#_Toc437374299)

[3.6.2 Forma de Cálculo 15](#_Toc437374300)

[4 Estudo Comparativo 16](#_Toc437374301)

[4.1 Estudo Algoritmos de Procura 16](#_Toc437374302)

[4.1.1 Critérios a analisar 16](#_Toc437374303)

[4.1.2 Testes Efectuados 17](#_Toc437374304)

[4.1.3 Resultados Obtidos 17](#_Toc437374305)

[4.1.4 Comparação dos Resultados Obtidos 18](#_Toc437374306)

[4.2 Estudo funções de custo/heurísticas 18](#_Toc437374307)

[4.2.1 Critérios a analisar 19](#_Toc437374308)

[4.2.2 Testes Efectuados 19](#_Toc437374309)

[4.2.3 Resultados Obtidos 19](#_Toc437374310)

[4.2.4 Comparação dos Resultados Obtidos 20](#_Toc437374311)

[4.3 Escolha da procura-best 20](#_Toc437374312)

# Implementação Tipo Tabuleiro e Funções do problema de Procura

## Tipo Abstracto de Informação Tabuleiro

Após a análise do problema concluímos que precisávamos de representar um tabuleiro de Tetris com 18 linhas e 10 colunas. Depois de investigarmos quais os tipos de estruturas nativos da linguagem Lisp ficámos com duas opções: lista e *array*.

As vantagens de escolhermos uma lista eram podermos utilizar funções recursivas, que proporcionam uma poupança de memória e facilidade de escrita de funções. As vantagens de utilizarmos um *array* são a facilidade de acesso a qualquer elemento através dos índices e utilizarmos funções iterativas que são mais eficientes a nível de tempo.

Ambas as opções eram perfeitamente possíveis de implementar, no entanto, ao analisarmos as funções necessárias para o projecto, verificámos que a facilidade de acesso no *array* era bastante importante pois na maior parte das funções é necessário aceder a posições arbitrárias no tabuleiro e este facto superava quaisquer benefícios que a lista pudesse trazer. Mais ainda, na função “tabuleiro-remove-linha!” iria ser benéfica uma implementação do tabuleiro na forma de lista pois seria trivial realizar esta operação, no entanto como esta operação é relativamente pouco utilizada em relação a todas as outras esta vantagem não seria relevante.

Assim sendo, e de forma a proporcionar uma menor complexidade ao nível da execução, decidimos escolher um *array* para implementar o nosso tipo de informação abstrata tabuleiro.

De forma sucinta, o tabuleiro é definido com um *array* bi-dimensional com 18 linhas por 10 colunas em que cada posição pode tomar o valor *true* ou *nil* o que corresponde a essa posição estar livre ou ocupada.

## Implementação de funções do problema de procura

### Função *solucao*

A função *solucao* recebe um estado e devolve o valor *true* se esse estado tiver a linha do topo livre e se a lista de peças por colocar estiver vazia, ou *nil* caso alguma das condições anteriores não se verifique.

Basicamente esta função é um predicado que diz se o estado representa uma solução do problema ou não.

### Função *accoes*

A função *accoes* recebe um estado e devolve uma lista de todas as acções válidas de acordo com a próxima peça a ser colocada e com o tabuleiro, ou seja, verifica qual é a primeira peça da lista de peças por colocar e para cada coluna e para cada orientação da peça constrói uma acção.

O primeiro passo do algoritmo é verificar se a lista de peças por colocar não se encontra vazia. Se a lista tiver vazia a função devolve imediatamente uma lista vazia, pois não existem quaisquer accoes possíveis.

Caso contrário, verificamos qual a próxima peça a colocar e criamos uma lista com todas as possibilidades de colocar essa peça no tabuleiro. De forma a simplificar o código usámos uma função auxiliar “cria-lista” que recebe a peça a colocar e o número de colunas em que essa peça pode ser colocada. Dado que a mesma peça pode ser colocada de forma diferente consoante a sua rotação, esta operação é repetida para todos as orientações possíveis da peça.

No final, é devolvida uma lista de todas as acções que são possíveis fazer no estado recebido.

### Função *resultado*

A função *resultado* recebe um estado inicial e uma acção, e devolve um estado sucessor que resulta de executar a acção no estado inicial.

De forma a explicar claramente o funcionamento desta função vamos fazê-lo por ordem sequencial:

1. Criação de um novo estado: É criado um novo estado igual ao anterior sendo que todas as próximas acções são executadas no novo estado, permanecendo o estado original inalterado.
2. Actualização das listas de peças: À lista de peças por colocar é retirado a primeira peça e esta é adicionada à lista de peças colocadas, no final da lista.
3. Colocação da peça no tabuleiro: Uso de uma função auxiliar *coloca-peca!* descrita abaixo.
4. Verificação do topo do tabuleiro: É verificado se o topo do tabuleiro está preenchido, caso esteja preenchido é devolvido o novo estado criado e não executada mais alteração nenhuma.
5. Linhas removidas e pontos: Caso o topo do tabuleiro não esteja preenchido, verificam-se quantas linhas estão totalmente preenchidas e as mesma são removidas, começando pelo topo do tabuleiro. Por último, são calculados os pontos consoante o número de linhas removidas.

### **Função *coloca-peca!***

A função *coloca-peca!* recebe um tabuleiro e uma acção e aplica essa acção no tabuleiro. Para calcularmos qual a posição onde inserir a peça no tabuleiro utilizamos a função altura-coluna para calcular qual a linha base onde a peça pode ser colocada (achando a altura máxima ao longo de todas as colunas ocupadas pela peça). Devido à irregularidade apresentada pelo formato das peças esta operação não é suficiente para determinar a linha de forma precisa sendo que precisamos ainda de verificar se a peça não se pode deslocar para baixo mais uma ou duas linhas. Para realizar esta última operação decrementamos a linha base caso se verifique que todas as casas abaixo da peça se encontram vazias, até uma delas não estar vazia.

Finalmente depois de sabermos qual a linha base (calculada previamente) e a coluna base (descrita na acção) percorremos o tabuleiro e a peça nas posições correspondentes e copiamos a peça para o tabuleiro.

# Implementação Algoritmos de Procura

Foram implementados 3 algoritmos de procura neste projecto. Dois desses algoritmos têm como objectivo tentar atingir, encontrar e devolver uma solução para o problema IA-Tetris, são a Procura em Profundidade e a procura A\*. O restante é a Tempera Simulada, um algoritmo usado para calcular coeficientes de forma a criar uma super-heurística, como será explicado mais à frente.

## Procura-pp

A procura em profundidade primeiro consiste num algoritmo de procura não informada que percorre a árvore de procura expandindo sempre o nó com maior profundidade, sendo que no caso de haver vários nós com a mesma profundidade se escolhe o mais à esquerda que não tenha sido expandido ainda. Para se obter este comportamento utiliza-se uma fila do tipo *last in first out* para guardar os nós não expandidos ou seja quando um nó é expandido os seus sucessores são colocados no final da fila e o próximo nó a ser expandido é sempre retirado do final da fila.

Para a implementação da fila LIFO decidimos utilizar uma lista pois é uma estrutura ideal para realizar as operações necessárias para uma fila deste tipo (*push* e *pop*). Optámos ainda por implementar a procura de forma iterativa pois, apesar de ser ligeiramente mais difícil de implementar oferece um melhor controlo sobre a execução do algoritmo e um melhor desempenho. Quanto à realização do teste objectivo este é feito durante a expansão mas uma possível optimização, que não nos ocorreu na altura, seria efectuar este teste na altura da geração o que iria permitir uma redução do número de nós explorados/expandidos.

Para conseguirmos calcular a lista de acções que levam o estado inicial ao estado objectivo necessitámos de para cada estado guardar o seu estado predecessor e a acção que ocorreu, para que fosse possível fazer o *backtracking* depois de encontrarmos a solução. Optámos por utilizar uma *hash-table* para armazenar esta informação pois é a estrutura de mapeamento mais eficiente que existe em lisp, também poderíamos ter utilizado uma *assoc list* porém o tempo de procura nesta lista seria muito superior.

Este tipo de procura não é óptima pois devolve a primeira solução que encontrar independentemente de esta ser ou não a melhor. Quanto à completude, como estamos a tratar de uma árvore de tamanho finito, a procura em profundidade é completa pois percorre todos os nós até encontrar uma solução, se esta existir.

Quanto à complexidade desta procura em relação ao tempo é O(bm) e em relação à memória é O(bm), sendo que b é o número de sucessores por nó e m a profundidade máxima da árvore de procura.

## Procura-A\*

A procura A\* consiste num algoritmo de procura informada que tenta minimizar o valor de uma função de custo estimado para encontrar uma solução óptima. A função de custo estimado é calculada a partir da soma do custo de caminho até ao estado actual (função de custo de caminho) e o custo estimado que falta para se chegar ao estado objectivo (função heurística).

Na implementação desta procura decidimos utilizar uma lista ordenada por valor da função de custo estimado para guardar o conjunto de estados gerados (conjunto de nós abertos/fronteira) sendo que a ordenação é feita no momento da inserção de um novo estado na lista. Decidimos fazer a ordenação no momento de inserção pois é mais eficiente do que percorrer sempre a lista toda quando se quisesse escolher o próximo nó para expandir. Quando ao conjunto de nós fechados não sentimos necessidade de guardar esta informação pois devido à natureza do nosso problema nunca existem estados repetidos, só faria sentido guardar esta informação para o caso de o algoritmo ser utilizado para fazer procuras em grafos com ciclos.

Novamente utilizámos uma *hash* *table* para guardar a informação necessária para calcular a lista de acções necessária para resolver o nosso problema.

Este tipo de procura, quando aplicado a uma árvore, é completo e é óptimo se a função heurística utilizada for admissível.

Quanto à complexidade esta depende da função heurística utilizada mas no pior caso é O(bd) sendo b o número de sucessores por nó e d a profundidade da solução.

## Têmpera simulada

Decidimos implementar a procura local têmpera simulada para calcularmos a melhor combinação de coeficientes na junção das várias heurísticas, decidimos optar por este tipo de procura por ser dos mais eficientes e fáceis de implementar dentro do conjunto de algoritmos descritos no livro da disciplina.

A procura têmpera simulada baseia-se no processo natural de têmpera de metais em que à medida que o metal vai arrefecendo a sua estrutura vai-se tornando mais refinada e estabilizando.

Transpondo esta metáfora para a computação em inteligência artificial a procura por têmpera simulada é uma procura que vai gerando estados de forma aleatória e passa de um estado para o outro se o novo estado obtiver uma avaliação superior ou, caso o novo estado seja pior que o estado actual, decide-se se este é aceite de acordo com a temperatura actual e com a diferença de avaliações, quanto maior for a temperatura maior e quanto menor for a diferença maior é a probabilidade de o novo estado ser aceite, sendo que a temperatura vai diminuindo ao longo da execução do algoritmo.

Para a implementação desta procura representámos um estado como sendo uma lista de coeficientes em que cada posição da lista guarda um coeficiente. Necessitámos de implementar um conjunto de funções auxiliares nomeadamente a função sucessor que recebe um estado e devolve um novo estado de forma aleatória, a função avaliação que realiza testes com os coeficientes aplicados na heurística e devolve uma pontuação que corresponde aos pontos obtidos e a função aceitação que devolve uma probabilidade de o novo estado ser aceite de acordo com a melhor pontuação obtida, a pontuação do estado actual e a temperatura.

# Funções Heurísticas

De forma a conseguirmos efectuar uma procura mais eficiente e obter o melhor resultado criámos diversas heurísticas que foram posteriormente testadas de forma a verificar até que ponto seriam ou não úteis para a procura. As heurísticas que desenvolvemos foram: Espaços livres totais, Linhas vazias, Colunas vazias, Espaços livres topo, Buracos totais, Média altura colunas, Linhas completas, Irregularidades, Altura colunas regiões e Bloco mais alto.

**Utilizadas: irregularidades, bloco-mais-alto, media-altura-colunas, linhas-vazias, buracos-totais, espaços-livres-totais**

## Espaços-livres-totais

### Motivação

Qualquer jogador de Tetris sabe que o melhor a fazer é ter o menor número de peças em jogo, ou seja, ter o máximo número possível de espaços livres no tabuleiro de jogo.

Sendo assim, esta heurística visa a dar prioridade a todos estados que possuam um menor número de “casas” ocupadas ao longo do tabuleiro para que sejam possíveis, posteriormente, mais jogadas e por conseguinte seja possível a obtenção de um número maior de pontos.

Esta heurística foi avaliada para diversas peças e vários tabuleiros, sendo que a curto prazo é uma heurística fundamental, uma vez que escolhe o estado que proporciona o sucesso imediato para uma dada peça.

### Forma de Cálculo

O cálculo desta heurística é trivial bastando percorrer todas as linhas e colunas contando o número de espaços em branco ao longo do tabuleiro e devolver o referido valor.

Exemplo de escolha de um estado:

Tabuleiro 1 Tabuleiro 2 Tabuleiro 3

Aplicando a heurística *espaços-livres-totais* aos estados dos tabuleiros 2 e 3, que resultam da aplicação da peça no tabuleiro 1, verificamos os seguintes cálculos:

* espaços-livres-totais(tabuleiro 2\*) = 177
* espaços-livres-totais(tabuleiro 3\*) = 167

Neste caso a heurística iria escolher o estado que tinha o tabuleiro 2 ao invés do tabuleiro 3, uma vez que o primeiro apresenta maior número de espaços livres totais o que iria proporcionar melhor probabilidade de atingir pontos no futuro dado que possibilitaria um maior número possível de jogadas.

\* - Estado que contém o tabuleiro n.

## – Linhas-vazias

### Motivação

Como vimos anteriormente, é importante manter o tabuleiro livre para jogadas futuras e aumentar a probabilidade de sucesso imediato, no entanto, como os pontos são obtidos quando se completa uma linha isto não chega para assegurar o máximo número de pontos.

A heurística linhas-vazias vem reforçar a ideia de que é essencial manter o tabuleiro “limpo”, desta forma, esta heurística beneficia tabuleiros que mantenham um maior número de linhas vazias em relação aos que têm tabuleiros que ocupem mais linhas, isto é, prejudica tabuleiros que sejam mais altos e com mais pontos desperdiçados.

Realizámos diversos testes, a cada heurística, um deles consistia em realizar uma procura A\* em 10 estados diferentes com dificuldade crescente criados aleatoriamente e para 4,5 e 6 peças sucessivamente, utilizando como estimativa do caminho a heurística a testar, no final analisámos os pontos obtidos pela procura. Encontram-se em baixo os resultados obtidos:

Verificámos que de uma forma geral a heurística obtém quase sempre pontos, ou seja, se jogássemos apenas com esta heurística podíamos ter alguma certeza que íamos ter boas escolhas e iríamos fazer pontos.

### Forma de Cálculo

Mais uma vez, e apesar de ser uma heurística fundamental e que proporciona bons resultados, esta é trivialmente calculada, bastando contar quantas linhas estão vazias, para tal, criámos uma função auxiliar *tabuleiro-linha-vazia-p* que recebe um inteiro correspondente à linha no tabuleiro e devolve *true* caso a mesma se encontre vazia e *nil* caso contrário.

Exemplo de escolha de um estado:

Tabuleiro 1 Tabuleiro 2 Tabuleiro 3

Aplicando a heurística linhas-vazias aos tabuleiros 2 e 3, que resultam da aplicação da peça no tabuleiro 1, verificamos os seguintes cálculos:

* linhas-vazias(tabuleiro 2) = 13
* linhas-vazias(tabuleiro 3) = 14

Podemos verificar que esta heurística é diferente e complementar da heurística descrita anteriormente, uma vez que aplicando somente a heurística *espaços-livres-totais*, a mesma não seria capaz de escolher entre o tabuleiro 2 e 3, dado que ambos têm o mesmo número de espaços livres, ao contrário desta heurística que tem a capacidade de decidir entre o tabuleiro 2 e 3 de forma clara.

## Buracos totais

### Motivação

Para além de marcar pontos é preciso assegurar que escolhemos um tabuleiro onde no futuro não seja impossível de marcar pontos, isto é, fazer jogadas que apesar de se pensar que reduzem os espaços livres ou as linhas vazias, não criem buracos que jamais poderão ser ocupados criando assim linhas mortas e indo acumulando peças ao longo do tempo vão aumentando a altura das colunas do tabuleiro.

Mais uma vez testámos esta heurística para 10 tabuleiros e sobre várias condições. Os resultados podem ser verificados no gráfico em baixo:

Como já era de esperar, esta heurística sozinha não tem como obter muitos pontos, no entanto, é importante para impedir que se percam pontos por ficarem buracos no interior do tabuleiro ou que se façam jogadas que comprometam o sucesso das próximas jogadas.

### Forma de Cálculo

Ao contrário das anteriores, o cálculo desta heurística não é trivial, primeiro é necessário termos noção do que é definido por um buraco. Na figura ao lado, as posições a vermelho representam buracos, ou seja, as linhas em que estes buracos aparecem só podem gerar pontos quando as linhas acima gerarem pontos. Para este caso, enquanto a linha 3 não gerar pontos, nem a linha 1 nem a linha 2 irão gerar pontos, fazendo com que o tabuleiro fique cada vez mais alto.

Sempre que encontramos, no tabuleiro, algo que satisfaça as condições para uma determinada posição ser considerada um buraco é incrementado um contador que é posteriormente devolvido.

Um buraco é definido com uma posição livre coberta por uma posição preenchida.

Exemplo da utilização da heurística:

Tabuleiro 1 Tabuleiro 2 Tabuleiro 3

Aplicando a heurística *buracos-totais* aos tabuleiros 2 e 3, que resultam da aplicação da peça no tabuleiro 1, verificamos os seguintes cálculos:

* buracos-totais(tabuleiro 2) = 2
* buracos-totais(tabuleiro 3) = 0

Sendo assim, concluímos que o tabuleiro 3 é preferido em relação ao tabuleiro 2, uma vez que apresenta um menor número total de buracos.

## Media-altura-colunas, altura-colunas-regiões

### Motivação

A heurística *media-altura-colunas*, actua de um modo superficial sobre um tabuleiro, permitindo escolher entre dois tabuleiros idênticos mas que se diferenciem na altura média das suas colunas.

Após a realização de testes, obtivemos os seguintes resultados:

Como seria de esperar, trata-se de uma heurística bastante poderosa, funcionando bem, mesmo sozinha, na decisão da colocação de peças num tabuleiro.

Para além desta heurística e atuando uma forma bastante semelhante mas com uma implementação mais complicada, temos a heurística *altura-colunas-regiões*. Esta heurística penaliza tabuleiros que se estendam na altura. Como se distingue das outras? Nesta heurística, a penalização é proporcional à altura do tabuleiro.

Na figura ao lado, é possível verificar as diferentes regiões que estabelecemos para o tabuleiro. Neste caso, esta heurística devolve a soma da altura das colunas multiplicada pelo coeficiente da região onde esta se encontra. Para tabuleiros que sejam muito inconstantes, esta heurística destaca-se em relação à *media-altura-colunas*.

### Forma de Cálculo

Em relação à heurística *media-altura-colunas*, o seu cálculo é obtido somando a altura de todas as colunas e dividindo pelo número total de colunas. Na *altura-colunas-regiões* o resultado é obtido somando a multiplicação da altura de cada coluna pelo coeficiente da região onde esta se encontra.

Tabuleiro 1 Tabuleiro 2 Tabuleiro 3

Aplicando a heurística buracos-totais aos tabuleiros 2 e 3, que resultam da aplicação da peça no tabuleiro 1, verificamos os seguintes cálculos:

|  |  |  |
| --- | --- | --- |
| Heurística/Tabuleiro | 2 | 3 |
| media-altura-colunas | 3.5 | 3.5 |
| altura-colunas-regiões | 240 | 113 |

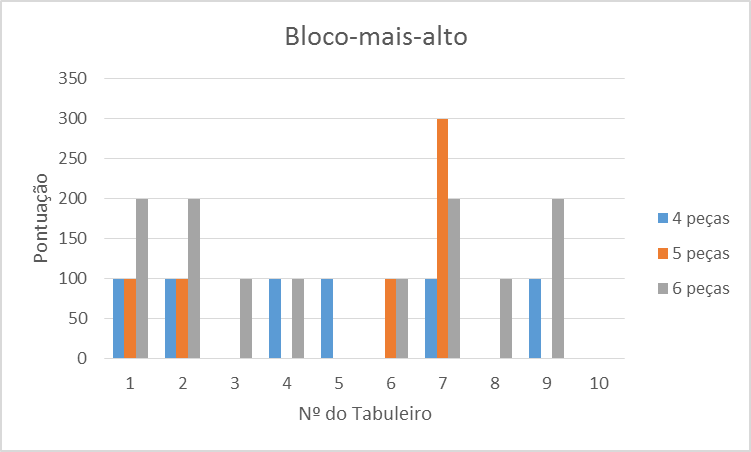
Neste caso, a media-altura-colunas é insignificante para escolher qual o melhor tabuleiro, no entanto, de um modo geral oferece um melhor desempenho como pode ser verificado pelo gráfico anterior. Uma vez que *media-altura-colunas* é relativamente mais fácil de calcular, é executada em menor tempo e oferece melhores resultados optámos por escolher esta heurística para integrar a nossa heurística final.

## Bloco-mais-alto

### Motivação

A heurística *bloco-mais-alto* pretende complementar a heurística media-altura-colunas pois a média não é suficiente para caracterizar um conjunto de resultados, pois consegue distinguir dois tabuleiros onde a média da altura das colunas seja igual. Para além disso foi uma das primeiras ideias que nos surgiu de forma intuitiva.

Após a realização de testes, obtivemos os seguintes resultados:



Apesar de ser uma heurística bastante simples os resultados foram surpreendentemente bons.

### Forma de Cálculo

Em relação à heurística bloco-mais-alto, o seu cálculo é muito simples bastando percorrer todas as colunas e ver qual a coluna mais alta, utilizando a função *altura-coluna.*

Tabuleiro 1 Tabuleiro 2 Tabuleiro 3

Aplicando a heurística bloco-mais-alto aos tabuleiros 2 e 3, que resultam da aplicação da peça no tabuleiro 1, verificamos os seguintes cálculos:

* bloco-mais-alto(tabuleiro2) = 10
* bloco-mais-alto(tabuleiro3) = 15

Neste caso seria escolhido o tabuleiro2 pois o valor da heurística é menor. É relevante notar que os tabuleiros 2 e 3 têm a mesma média de altura das colunas, mostrando assim como a heurística *bloco-mais-alto* complementa a heurística *media-altura-colunas*.

Esta heurística surgiu de forma trivial e intuitiva, não tendo sido necessário qualquer processo de depuração. Como esta heurística apresenta uma boa sinergia com a *media-altura-colunas*, é bastante eficaz e fácil de calcular decidimos incluí-la na heurística final.

## Irregularidades

### Motivação

A heurística *irregularidades* foi baseada na ideia de que um tabuleiro plano oferece um maior número de jogadas possíveis e um menor desperdício de pontos, pois num tabuleiro plano qualquer peça pode ser colocada em qualquer posição enquanto que num tabuleiro muito irregular pode ser complicado encaixar determinadas peças.

Após a realização de testes, obtivemos os seguintes resultados:

Como seria de esperar, o número de pontos ganhos não foi muito alto pois o objectivo desta heurística é manter o tabuleiro num estado propício para que se possam fazer pontos e ser utilizada em conjunto com outras heurísticas.

### Forma de Cálculo

Em relação à heurística *irregularidades*, o seu valor é obtido somando o valor absoluto da diferença de alturas entre todos os pares de colunas adjacentes ou seja quanto mais irregular for o tabuleiro maior é o resultado da heurística e num tabuleiro totalmente plano o valor da heurística é igual a 0.

Exemplo da utilização da heurística:

Tabuleiro 1 Tabuleiro 2 Tabuleiro 3

Aplicando a heurística *irregularidades* aos tabuleiros 2 e 3, que resultam da aplicação da peça no tabuleiro 1, verificamos os seguintes cálculos:

* irregularidades(tabuleiro2) = |1-1|+|1-1|+|1-1|+|1-0|+|0-0|+|0-2|+|2-4|+|4-0|+|0-0| = 9
* irregularidades(tabuleiro3) = |1-1|+|1-1|+|1-1|+|1-0|+|0-0|+|0-1|+|1-1|+|1-1|+|1-3| = 5

A heurística *irregularidades* iria escolher o tabuleiro 3 porque este é mais plano e é mais fácil encaixar novas peças no tabuleiro 3 do que no tabuleiro 2.

Decidimos incluir esta heurística na nossa heurística final pois ajuda a manter um tabuleiro organizado tal como a heurística *buracos-totais.*

# Estudo Comparativo

Na *secção 3* é apresentado, de forma clara e detalhada uma explicação sobre todas as heurísticas implementadas, bem como das heurísticas utilizadas para criar a *Heurística-Final.* Esta heurística foi criada através da junção das 6 melhores heurísticas: *irregularidades, bloco-mais-alto, media-altura-colunas, linhas-vazias, buracos-totais, espaços-livres-totais.*

## Estudo Algoritmos de Procura

Ao longo da realização do projecto implementámos dois algoritmos de procura para resolver o IA-Tetris, a procura A\* e a procura em profundidade. Para além destes algoritmos implementámos também dois algoritmos locais: têmpera simulada e uma aproximação de *random-hill-climbing*. Estes dois algoritmos serviram para calcular os coeficientes que iriamos atribuir posteriormente a cada heurística por forma a criar uma heurística final, a ***Heurística-Final***.

Uma vez que ambas as procuras já foram descritas anteriormente não nos iremos estender novamente na descrição das mesmas, sendo assim, e de uma forma reduzida: *Procura em Profundidade*, é uma procura que dado um problema, se existir uma solução esta é encontrada, no entanto, para problemas que têm mais que uma solução não é garantido que a melhor solução seja a solução encontrada; procura A\*, é uma procura informada, ou seja, usa informação resultante de heurística de forma a conduzir o caminho para uma solução, como o nosso problema é representado por uma árvore de procura se a heurística for admissível, então a solução encontrada é a melhor. Para o problema IA-Tetris, não podemos garantir a 100% que as heurísticas tenham esta definição, no entanto, as heurísticas foram combinadas de forma a produzir o resultado mais viável possível.

### Critérios a analisar

De forma a comparar as duas procuras, criámos uma função *teste* que cria tabuleiros aleatórios e um conjunto de peças aleatórias e devolve um ficheiro onde indica para cada tabuleiro quantos pontos a procura conseguiu atingir. Sendo assim, considerámos que a obtenção de **pontos** por cada procura era essencial para definirmos qual das procuras usar.

No entanto, não é só os pontos que interessam, não vale ter uma boa procura se a mesma demorar muito tempo a executar, sendo assim consideramos também que o **tempo de execução** é um critério extremamente importante.

Por último, não nos serve de nada termos uma procura que apesar de fazer rapidamente muitos pontos, gasta tanta memória naquele espaço de tempo que a memória não é proporcional à quantidade de peças que são inseridas no tabuleiro. Sendo assim, a **quantidade de peças** que a procura aguenta foi o último dos critérios a analisar.

### Testes Efectuados

Foram efectuados diversos testes consoante os critérios, para o critério:

* Pontos – Corremos ambas as procuras, alternadamente no mesmo tabuleiro para 20 tabuleiros diferentes, registámos o número de pontos que cada uma conseguiu atingir.
* Quantidade de peças (memória) – Verificamos se, dado dois tabuleiros e fazendo variar o número de peças, qual seria o número máximo de peças que cada procura conseguia processar e devolver uma solução.

### Resultados Obtidos

Para o critério **Pontos** obtivemos os seguintes resultados:

Para o critério **Quantidade de Peças** obtivemos os seguintes resultados:

|  |  |  |
| --- | --- | --- |
| **Procura** | **Número Máximo de Peças** | |
| **Tabuleiro vazio** | **Tabuleiro meio cheio** |
| **Procura em Profundidade** | 50 | 50 |
| **A\*** | 4 | 3 |

### Comparação dos Resultados Obtidos

Para o critério **Pontos**:

A vitória é clara: A\*. A procura **A\*** consegue realizar pontos em quase todos os tabuleiros 18/20, a procura em profundidade fica pelos 2/20.

Para o critério **Quantidade de Peças:**

A *procura-pp* consegue achar uma solução para uma quantidade impressionante de 50 peças enquanto que a *procura-A\** apenas consegue 3-4 peças.

## Estudo funções de custo/heurísticas

Como já referimos anteriormente, criámos uma super-heurística, a *Super-Heurística*. Esta resulta da junção de 6 heurísticas: *irregularidades, bloco-mais-alto, media-altura-colunas, linhas-vazias, buracos-totais, espaços-livres-totais.* Estas heurísticas foram as que se destacaram nas análises individuais efectuadas.

*Mas como combinar estas heurísticas tão diferentes numa só?* Este foi o nosso principal problema, poderíamos simplesmente devolver uma soma dos resultados de cada uma destas heurísticas, no entanto essa não seria certamente a melhor opção.

Como optimizar um conjunto de heurísticas numa só? Um verdadeiro problema de optimização, e para resolver este problema implementámos o algoritmo **Tempera Simulada.** Atribuímos a cada heurística um coeficiente que seria multiplicado por essa heurística, e devolvíamos a soma destas multiplicações. O Resultado final é a ***Super-Heurística*.**

**Super-Heurística = C1.***irregularidades* **+ C2.***bloco-mais-alto* **+ C3.***media-altura-colunas* **+ C4.***linhas-vazias* **+ C5.***buracos-totais* **+ C6.***espaços-livres-totais*

Usando a têmpera simulada e um *random-hill-climbing* obtivemos os valores para os coeficientes C1 a C6.

### Critérios a analisar

*Como usar estes algoritmos para achar os valores perfeitos para os coeficientes?*

Gerar os coeficientes que, dado um tabuleiro e um conjunto de peças, consigam obter o maior número de **pontos** para cada jogo.

### Testes Efectuados

Criar um tabuleiro e um conjunto de peças, definir um número máximo de pontos que podem ser atingidos, este passo foi feito manualmente por nós, como jogadores. De seguida, correr os algoritmos até que estes devolvessem coeficientes que cumprissem o objetivos que lhes demos.

Neste caso estipulámos um tabuleiro em que eram possíveis atingir 600 Pontos, com as 4 peças que faltavam colocar.

Este teste é a melhor opção para combinar 6 coeficientes, uma vez que se trata de um problema de optimização nada melhor que a procura local para resolver o problema ao invés da tentativa erro que poderia ser bastante morosa.

### Resultados Obtidos

Este gráfico demonstra uma dispersão dos valores do coeficientes que permitem atingir 400 Pontos, o número de pontos máximos que conseguimos atingir, fazendo variar os coeficientes entre -10 e 10.

### Comparação dos Resultados Obtidos

Após a análise do gráfico de dispersão verificamos que os valores ideais são:

**Super-Heurística = 1.6028517.***irregularidades* **– 2.0820208.***bloco-mais-alto* **-0.665044406.***media-altura-colunas* **+ 0.7125199.***linhas-vazias* **+ 8.79986.***buracos-totais* **+ 4.750551.***espaços-livres-totais*

## Escolha da procura-best

Com base em todos os testes anteriores desde a secção 3 até à 4.2 deste documento a nossa escolha foi implementar a procura best como uma procura A\* e como heurística que estima a qualidade do caminho para atingir uma solução usámos a já denominada **Super-heuristica.**

A procura A\* apesar de consumir uma grande quantidade de memória, pela quantidade de nós que ficam guardados em memória é, em relação à procura em profundidade, uma clara vencedora uma vez que é capaz de produzir uma verdadeira solução de jogo, isto é, é capaz de criar uma solução que gere pontos, sempre que seja possível.

**NOTA SOBRE O PROJECTO/CÓDIGO ENTREGUE**

Como é possível verificar pelo projecto entregue, bem como pelo relatório, o nosso esforço em desenvolver um óptimo projecto foi extenso e árduo, no entanto, não conseguimos atingir o sucesso nos testes automáticos. Até à data da entrega *online*, fizemos tudo o que era possível para perceber que erro tínhamos no projecto, infelizmente, não conseguimos.

Enquanto realizávamos o relatório tivemos a ideia de deixar que os coeficientes da ***Super-heuristica*** variassem entre 0 e 100 ao invés de -10 e 10, como é possível verificar nos testes anteriormente efectuados. Tal foi o nosso espanto quando percebemos que conseguíamos agora atingir os pontos que estipulámos nos objectivos.

Apesar de tudo, foi enviada uma versão corrigida do projecto que gostávamos, dando o empenho que tivemos, que fosse tida em conta aquando da avaliação do projecto.