Grupo Nº 35

TÉCNICO LISBOA

Inteligência Artificial

1.º Semestre 2015/2016

IA-Tetris

Relatório de Projeto

Índice

1 Implementação Tipo Tabuleiro e Funções do problema de Procura

1.1 Tipo Abstrato de Informação Tabuleiro

O nosso tabuleiro é representado diretamente por um *array* bidimensional (pode-se também dizer que consiste numa matriz 18 por 10), constituído por outros 18 *arrays* em que cada um destes tem um tamanho de 10 elementos (do tipo booleano, isto é, assumindo os valores *t(true)* ou *nil*). Foi escolhida esta representação pelo facto de este tipo de informação apenas necessitar de guardar a informação relativa a cada uma das células constituintes do tabuleiro (sendo que esta informação apenas pode assumir dois valores: ou a célula está ocupada ou, pelo contrário, está livre). Como seria de esperar, existiam outras possibilidades em aberto para a representação do tabuleiro e por isso tivemos de pesar as vantagens e desvantagens de cada uma para decidir qual a melhor para todas as operações que teríamos de realizar sobre este tipo:

Poderíamos, por exemplo, criar uma estrutura própria para o tabuleiro, mas concordámos que esta escolha só se justificava se possuíssemos uma maior quantidade de informação a ser armazenada na estrutura. Isto não acontece neste caso já que apenas temos de guardar o estado de cada célula do tabuleiro como já foi explicado no primeiro parágrafo.

Havia ainda outra opção, considerada mais viável em comparação com a que acabámos de descrever, que seria representar este tipo de informação com uma lista constituída por outras listas. Desta maneira, seria algo bastante parecido com o que implementámos, mas com a diferença subtil de que, em vez do uso de *arrays*, usaríamos listas associadas as linhas do tabuleiro. Não seguimos em frente com esta opção porque percebemos, olhando para todas as funções que teríamos que implementar relacionadas com tabuleiros, que não tínhamos apenas simples consultas (por exemplo, *tabuleiro-topo-preenchido-p*) mas também operações que implicavam a sua modificação (por exemplo, *tabuleiro-preenche!*). Sendo assim, a conclusão a que o grupo chegou foi que as listas, apesar de permitirem consultas sem problemas, eram um caso completamente diferente no que toca a alterações ao seu estado pois apresentavam uma maior dificuldade, sobretudo em comparação com a facilidade que encontrámos a modificar elementos de *arrays*. Por outro lado, funções como transformar tabuleiros em *array* (*tabuleiro->array*), e vice-versa (*array->tabuleiro*), foram também elas uma forte motivação para a decisão tomada quanto à representação de um tabuleiro.

Explicado todo o processo de escolha da implementação deste tipo, passemos para a sua descrição. Os 18 *arrays* correspondem às linhas do tabuleiro do jogo, sendo que o *array* 0 é a linha do topo (parece contraditório, mas acaba por ajudar na visualização do tabuleiro na altura do output) e o *array* 17 a base do tabuleiro. Os 10 elementos que constituem cada um dos 18 *arrays* representam as 10 colunas/células que cada linha do tabuleiro possui, sendo o elemento 0 a coluna mais à esquerda e o elemento 10 a coluna mais à direita. Para representar se essas posições/células do tabuleiro estão livres ou ocupadas, usamos as variáveis booleanas *nil* e *t(true)*, respetivamente. Notar que, na criação de um novo tabuleiro (usando a função *cria-tabuleiro*), este é inicializado completamente vazio, isto é, todas as células vão começar com valor *nil*.

Achámos por bem falar um pouco sobre a implementação das funções pedidas no enunciado (na altura para a 1ªEntrega do projeto), mencionando dificuldades/facilidades devido à representação escolhida para o tipo abstrato de tabuleiro.

A verificação do estado de uma posição é realizada através da função *tabuleiro-preenchido-p*, passando lhe um número correspondente à linha e outro correspondente à coluna da célula que queremos analisar (recurso à função *aref*, pertencente à biblioteca do *Clisp*).

A operação de copiar tabuleiros acabou por se revelar um pouco mais difícil já que nos "obrigou" a construir uma função auxiliar *2d-array-to-list* para transformar o *array* correspondente ao tabuleiro recebido numa lista sem qualquer perda de informação. Desta maneira, conseguimos criar um tabuleiro com o mesmo estado que o que a função recebeu e devolver uma autêntica cópia, completamente independente do tabuleiro original (qualquer alteração não irá ser refletida neste).

Em relação ao resto das funções foi relativamente fácil de desenvolvê-las pois apenas envolviam simples verificações do estado do tabuleiro, nada que tenha apresentado grandes obstáculos... Destaque para as duas funções *tabuleiro->array* e *array->tabuleiro*, que na verdade fazem a mesma coisa (reaproveitam o código), pois devido à representação do tabuleiro na forma de um *array* bidimensional, a única coisa a fazer é mesmo inverter as linhas do tabuleiro (assim, a primeira linha do *array* recebido/devolvido vai corresponder à base do tabuleiro enquanto a última está associada ao topo).

1.2 Implementação de funções do problema de procura

1.2.1 Função *soluçao*

Comecemos então pela função solução que, como se pode entender pelo nome, permite identificar um estado como sendo solução do problema (neste caso falamos especificamente do problema do Tetris). Esta função foi particularmente fácil de escrever já que possuímos duas funções auxiliares que nos ajudaram.

Desta maneira, o valor retornado é o resultado de uma condição lógica "AND" na qual o tabuleiro do estado recebido não pode ter o topo preenchido (chamada à função tabuleiro-topo-preenchido-p) e deve ser um estado final (chamada à função estado-final-p que verifica se o topo do tabuleiro está preenchido ou não existem peças para colocar nesse estado).

Concluindo, a condição lógica devolve um valor *true* se o estado não tiver um tabuleiro com topo preenchido e todas as peças já foram colocadas (este acaba por ser a definição da solução do nosso problema) e *nil* caso contrário.

1.2.2 Função *accoes*

Seguimos para a função *accoes* que nos vai devolver uma lista de ações possíveis fisicamente para a peça que queremos colocar. Sendo assim, começa por verificar o símbolo correspondente à próxima peça a colocar no tabuleiro (para isso, consulta a lista de pecas por colocar do estado recebido). Durante este passo achámos por bem desenvolver uma função simbolo->pecas com o intuito de receber a lista de configurações possíveis da peca correspondente ao símbolo que lhe passamos como argumento (estas configurações encontram-se no ficheiro *utils.lisp*).

Possuindo esta lista a função precisa agora de chegar às ações possíveis para cada "rotação" da peca. Iterando sobre a lista e consultando a dimensão horizontal de cada configuração, vamos adicionando à lista de ações possíveis (que vai ser retornada pela função) começando pela coluna mais à esquerda no tabuleiro, 0, até à ultima coluna que, adicionando o comprimento horizontal da peça, não ultrapasse os limites do tabuleiro. Assim, no final temos todas as ações para todas as configurações da peca a colocar que sejam fisicamente viáveis de efetuar dentro do tabuleiro.

É importante referir que, no caso do topo do tabuleiro estiver preenchido, esta função devolve nil, bem como se não houver mais pecas para colocar.

1.2.3 Função *resultado*

Falamos agora da função mais complicada de implementar entre estas que estamos a tratar: função resultado.

Em primeiro lugar, copiamos o tabuleiro do estado recebido para uma nova variável e recorremos a uma função auxiliar coloca-peca (que explicamos de seguida), que é responsável pela colocação correta da peca. Obtendo o tabuleiro com a peca colocada procedemos à verificação das linhas completas (isto no caso de o topo não ficar preenchido depois da ação tomada).

Assim podemos calcular os pontos conseguidos com a jogada e podemos criar o estado resultante que é devolvido pela função. Este é constituído pelo tabuleiro já com a peca colocada e linhas completas removidas, pela soma dos pontos ganhos com os pontos que tínhamos anteriormente, e pela deslocação do símbolo da peca colocada da lista de pecas-por-colocar para as pecas-colocadas.

1.2.4 Função *coloca-peca* (função auxiliar)

A simplificação da função anterior apenas foi possível através da criação da função coloca-peca com o objetivo de abstrair a função resultado de todo o processo adjacente à colocação da peca num tabuleiro (já preenchido ou não) na posição correta de acordo com a ação recebida.

Apesar de parecer fácil na teoria, acaba por ser mais complicada de implementar pois envolve a verificação do estado atual do tabuleiro e se a colocação da peça numa certa posição não vai sobrepor células já ocupadas ou tapar um "buraco" (movimento igualmente rejeitado neste jogo).

Assim sendo, a função executa um ciclo (que termina quando encontrar a primeira posição possível) em que procura as coordenadas verticais da peça (falamos a nível vertical pois as coordenadas horizontais são dadas na ação – numero da coluna mais a esquerda), confirmando se as condições anteriormente referidas não se verificam para qualquer célula ocupada pela peça: aceitando a posição se assim for, ou subindo uma linha no tabuleiro e repetindo o processo. No final, a função devolve o tabuleiro com a peça nas coordenadas corretas.

1.2.5 Função *custo-oportunidade*

Por fim, a última função implementada deste conjunto de funções do problema de procura foi a função de custo-oportunidade que, basicamente, nos revela qual a diferença entre os pontos que obtivemos até ao estado atual e a pontuação máxima que seria possível com as peças já colocadas (desta maneira, considera-se uma medida para ajudar na escolha entre estados a explorar por parte do algoritmo e procura).

2 Implementação Algoritmos de Procura

2.1 Procura-pp

Depois de estudada a procura em profundidade nas aulas teóricas, foi pedido para desenvolver durante este projeto, um algoritmo que implementasse essa mesma procura e resolvesse não só o problema do *Tetris*, como outros. Resumidamente, este vai, como o nome indica, procurar um estado solução, em profundidade primeiro, ou seja, desce no grafo de nós, só voltando para trás se nenhuma solução for encontrada.

A implementação não foi algo de complicado, tratando-se sobretudo de uma correta atualização de variáveis ao longo da procura. Ainda assim, foi necessário a criação de uma estrutura auxiliar (não só para este algoritmo de procura como o próximo) estado-f que tem como atributos uma referência para o próprio estado, o valor atribuído pela função heurística (não importante neste caso) e as ações tomadas desde o estado inicial do problema para se ter chegado a este estado (e isto sim, é o contributo mais importante por parte desta estrutura).

Desta maneira, temos uma lista de estados por explorar (*lista-estados-por-explorar*) que possui em todo e qualquer momento, todos os nós que o algoritmo tem por explorar e, neste caso, encontra-se ordenado por ordem de adição à lista (sem que exista uma função de ordenação pois não é necessária). Temos a *lista-acoes-tomadas* que nos diz, para o estado que estamos a explorar no momento, quais as ações que levaram a este e permite transmiti-la aos estados resultantes. Esta lista também pode ser vista como uma lista dos nós já explorados até ao momento.

Posto isto o algoritmo entra num ciclo enquanto não chegar a um estado que constitua uma solução do problema recebido ou este não tenha um estado com essa condição (neste caso, o algoritmo explora todos os estados possíveis à procura). É neste ciclo principal que vamos atualizando o estado-atual com o último estado a ser adicionado à lista de estados-por-explorar, bem como todas as outras variáveis.

Em cada iteração, exploramos o estado atual, adicionando à fronteira de nós a explorar todos os estados que resultam das ações possíveis a partir desse mesmo estado. Neste ponto, adicionámos uma medida de otimização que vista diminuir o número de nós explorados: não vale o esforço adicionar estados que não possuam ações possíveis e não constituem uma solução do problema, que, significaria que são um “beco sem saída” e como tal, nem é necessário explorar (não vai contribuir de qualquer maneira).

Quando for encontrada uma solução, o algoritmo sai imediatamente do ciclo principal e devolve as ações tomadas até ao estado atual que neste caso é solução, ou seja, todas as peças foram colocadas e o topo do tabuleiro não foi preenchido. A pouco complexidade na implementação deste algoritmo deve-se ao facto de não ser preciso manipulação da fronteira sem ser fazer operação de “pop” para retirar o primeiro elemento (o mais recente e mais “profundo”).

2.2 Procura-A*

A implementação do algoritmo de procura A* por parte do nosso grupo tem muito que se diga e se explique, como veremos nesta secção.

Tal como no algoritmo anterior, muitos processos foram facilitados através da estrutura estado-f. Assim, possuímos uma lista com todos os nós abertos, ou seja, estados por explorar, sendo que os seus constituintes são da estrutura já referida estado-f, isto porque simplifica o processo de procura do estado com menor valor para explorar.

Temos mais duas listas: uma com as ações possíveis através do estado-atual (variável correspondente ao nó a ser explorado na presente iteração) que vai permitir a atualização da fronteira; a outra é a lista de ações tomadas que pode ser considerada uma espécie de lista de nós fechados/explorados bem como um histórico das decisões tomadas pelo algoritmo até ao momento e que é devolvida no final.

Depois da inicialização de todas as variáveis, o algoritmo entra num ciclo do qual apenas sai se encontrar uma solução ou a fronteira estiver vazia (lembramos que esta é criada já incluindo o estado inicial do problema). Dentro deste ciclo, efetuamos a atualização da fronteira, analisando as ações possíveis a partir do estado atual e adicionando os estados resultantes (chamada a função resultado) à lista de estados por explorar.

Feito isto, estando nós a falar do algoritmo de procura A*, é necessário procurar o estado com o menor valor entre toda a lista de nós abertos, operação cuja responsabilidade recai sobre a função auxiliar valor-mínimo (vai correr toda a lista na busca do estado cm menor valor). Assim fica apenas a faltar a atualização de todas as variáveis antes de passar para a próxima iteração do ciclo principal.

Falta então referir as duas medidas de otimização tomadas pelo nosso grupo. Sabendo que a eficiência do algoritmo (tanto no aspeto temporal como espacial) depende do tamanho da lista de estados por explorar (fronteira), os nossos esforços para otimizá-lo dirigiram-se sobretudo para esse objetivo de reduzir a fronteira.

A primeira medida consiste em apenas adicionar estados à fronteira que não são estados finais ou, sendo estado final, é uma solução do problema. Desta maneira, poupamos o esforço de explorar estados que, na verdade, não vão ajudar a chegar ao nosso objetivo.

A outra medida consiste em limitar os nós a adicionar à fronteira, impondo uma constrição nos valores que assumem. Na nossa opinião, é uma decisão bastante discutível e errada em muitos aspetos já que acaba por alterar drasticamente o algoritmo. Isto porque, apesar de ser benéfico em certos casos/testes/problemas, noutros acaba por ter um desempenho algo desastroso pois impede o algoritmo de chegar a uma solução do problema mesmo que esta seja trivial de alcançar (por exemplo, colocar uma peça num tabuleiro já com diversos “buracos” por preencher). Sendo esta a secção de descrição do algoritmo, ficamos por aqui, deixando a restante análise de desempenho para secções posteriores do relatório (Estudo Comparativo).

3 Funções Heurísticas

3.1 Qualidade

3.1.1 Motivação

Sendo o objetivo do jogo obter o maior número de pontos possível, o nosso grupo utilizou esta heurística que permite-nos saber exatamente o valor que certo estado representa a nível de pontos alcançados até ao momento. Sabendo que procuram o estado que tenha o maior número de pontos, é óbvio que queremos dar preferência aos estados que caminham para essa mesma solução. Para esta heurística, foi usada uma componente da estrutura estado, os pontos.

3.1.2 Forma de Cálculo

No cálculo desta heurística, foram utilizados os pontos de um estado, e foi calculado o simétrico desse valor, para que, aquando da utilização da heurística por parte de um algoritmo (que irá calcular o estado com menor valor entre a lista de estados por explorar) esse valor favoreça estados com maior número de pontos.

3.2 Buracos

3.2.1 Motivação

Pode-se considerar senso comum que um dos objetivos secundários do jogo de *Tetris* é provocar o menor número de "buracos" no tabuleiro com as ações que tomamos. Isto deve-se ao facto de que, a partir do momento em que um buraco foi criado com a colocação de uma peça de determinada forma, somos obrigados a preencher as linhas que estão a "tapá-lo" (todas as células ocupadas na mesma coluna que o buraco em questão e situadas em cima deste). Para tal, é necessário um esforço adicional que queremos evitar.

Como a verdade é que criando um buraco, existem grandes probabilidades de a situação do tabuleiro piorar, ao contrário de se resolver, tornou-se óbvio para nós que estava aqui uma heurística fundamental para o nosso projeto. Podemos mesmo dizer que a performance do jogador desce com o número de buracos que possui no tabuleiro. Para o cálculo desta heurística, foi usado o tabuleiro que se encontra dentro da estrutura *estado*.

3.2.2 Forma de Cálculo

Para calcular o número de buracos existentes no tabuleiro é necessário percorrer todas as colunas de cima para baixo, verificando se alguma célula está ocupada. A partir do momento em que encontra uma célula ocupada numa coluna, todas as células livres com que se deparar na mesma, são consideradas "buracos". Tudo isto foi calculado através de variáveis booleanos atualizadas durante os ciclos de verificação.

3.3 Higher Slope & Total Slope

3.3.1 Motivação

Outro dos aspetos que o nosso grupo reparou que contribuía para um aumento de performance por parte do jogador era a "uniformidade" de alturas que se mantem no tabuleiro enquanto se coloca as pecas. Podemos mesmo dizer que se procura ter a menor altura possível no tabuleiro.

Isto tem a sua razão de ser pois uma coluna que apresente uma altura elevada significa que precisaremos de eliminar um maior número de linhas para "apagar" o rasto da peça que acabámos de colocar. Outra maneira de observar isto é que quanto mais alta for uma coluna, mais perto o jogador se encontra da linha do topo e como tal, aproxima-se da situação do *game-over*, contrário a tudo aquilo que queremos que o algoritmo faça.

Assim pensámos em escrever duas heurísticas: *Higher Slope* e *Total Slope* que, apesar de usarem a mesma ideia como base, avaliam a mesma situação de maneira diferente. O *Higher Slope* aponta para situações em que se criou uma coluna com uma altura significativa em relação a todas as outras, sendo que, por outro lado, o *Total Slope* olha mais para a já referida uniformidade de alturas das colunas do tabuleiro e a variação que existe entre todas estas. Só os testes a serem realizados à frente nos dirão se alguma destas merece ser utilizada ou mesmo as duas, e com que importância no valor final de um estado. Para estas duas heurísticas, foi usado o tabuleiro que se encontra dentro da estrutura *estado*.

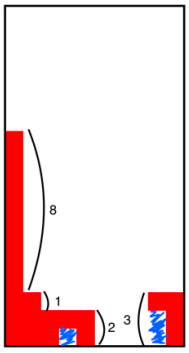
3.3.2 Forma de Cálculo

O *slope* de uma coluna é calculado através da seguinte expressão:

$$Slope\ da\ Coluna\ [i] = \left| Altura\ da\ Coluna\ [\ i\ ] - Altura\ da\ Coluna\ [\ i+1\ ] \right|$$

Sendo que o *Total Slope* consiste na soma deste calculo para todas as colunas entre 0 e 8 enquanto o *Higher Slope* corresponde ao *slope* máximo entre todas as colunas.

$$Total\ Slope = \sum_{i=0}^{i=8} Slope\ da\ Coluna\ [\ i\ ]$$



Neste exemplo, podemos observar que o slope da coluna 0 é 8, da coluna 1 é 1, da coluna 4 é 2, da coluna 7 é 3. Assim, o higher-slope é 8 e o total-slopes é 8+1+2+3=14. O número de buracos é 3 e os espaços preenchidos são 23.

3.4 Espaços Preenchidos

3.4.1 Motivação

Por fim, como última heurística que o nosso grupo decidiu implementar no projeto, temos o número de espaços preenchidos no tabuleiro. O nosso objetivo, ao criar esta heurística, foi tentar que o jogador impedisse que, à medida que o tabuleiro fosse preenchido, este ficasse "muito cheio". Isto porque, quanto mais cheio o tabuleiro tiver, mais difícil vai ser fazer as jogadas de modo a tentar obter o maior número de pontos. É claro que isto só se justifica a partir de um certo número de linhas, até porque, ao fazer 4 linhas, é recebida a maior pontuação numa jogada. Para esta heurística, foi usado um tabuleiro que se encontra na estrutura *estado*.

3.4.2 Forma de Cálculo

Para calcular os espaços preenchidos de um tabuleiro é necessário percorrer todas as linhas e todas as colunas, verificando se alguma posição está ocupada. Se estiver incrementa o número de espaços preenchidos.

4 Estudo Comparativo

4.1 Estudo Algoritmos de Procura

4.1.1 Critérios a analisar

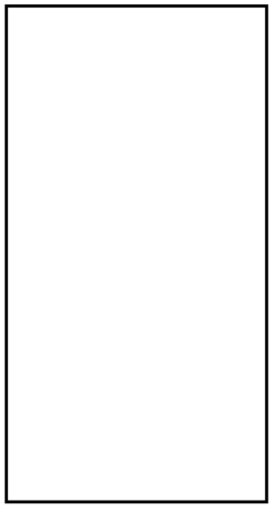
O nosso grupo selecionou um determinado conjunto de critérios de modo a ajudar a avaliação e comparação entre os vários algoritmos implementados. A escolha dos critérios foi feita de modo a abranger o máximo de fatores possíveis que permitissem melhor deteção de erros no desenvolvimento dos algoritmos e uma comparação mais completa entre eles. Os critérios escolhidos foram então:

- **Qualidade das Jogadas:** critério ligado sobretudo à pontuação final atingida pelo algoritmo, mas também as pontuações intermédias conseguidas. Desta maneira, é analisado se, entre as várias ações possíveis de acordo com o estado atual, o algoritmo procura uma jogada que contribua para o objetivo de obter o máximo de pontos possíveis tendo em conta o tabuleiro e a lista de pecas por colocar que constituem o estado inicial do problema recebido.
- **Tempo de Execução:** uma das maiores preocupações na elaboração/escrita dos algoritmos foi a sua complexidade temporal, isto é, o tempo que demorava a sua execução completa. Este critério ajuda a compreender quais os custos a nível de tempo associados a cada um e, também, como estes variam de acordo com o tamanho dos parâmetros que recebem.
- **Nós/Estados Visitados:** apesar de estar de certa forma ligado ao tempo de execução, era, na nossa opinião, importante saber quantos nós são explorados. Assim conseguimos ter uma ideia do número de "decisões" tomadas pelo algoritmo, não esquecendo (ver secção sobre a implementação do algoritmo se for necessário) que cada uma destas corresponde a uma iteração do ciclo principal da função com todas as consequências temporais e espaciais associadas. Outro motivo que nos levou a incluir este critério foi o facto de nos permitir avaliar, de forma indireta, a complexidade espacial dos algoritmos. Acontece que a memória alocada durante a execução depende do número de nós explorados porque quantos mais estados visitados, mais estados serão adicionados à lista de estados por explorar.
- **Número Máximo de Jogadas:** para além do tempo e memória, era fundamental perceber que limite (de certa maneira, impeditivo) tínhamos para cada algoritmo a nível de jogadas efetuadas. Neste critério foi necessária uma atenção redobrada porque o número máximo de jogadas pode estar associado a diversos aspetos do problema recebido e por isso, foi preciso perceber as razões da natureza destes números calculados para os algoritmos.
- **Estado do Tabuleiro Recebido:** este critério foi, propositadamente, adicionado pelo nosso grupo de forma a apontar um dos maiores defeitos da nossa implementação do algoritmo de procura A* como veremos na secção dos resultados e comparação. A verdade é que todos os problemas recebidos pelos algoritmos são diferentes à sua maneira e um dos maiores motivos para que isso aconteça é o estado do tabuleiro ligado ao estado inicial, que vai pesar significativamente no nível de dificuldade do problema.
- **Solução Encontrada:** para completar a listagem de defeitos do algoritmo de procura A* por nós implementado, submetemos este critério que avalia se o algoritmo conseguiu chegar a uma solução depois da sua execução completa (relembramos que uma solução deste problema do *Tetris* consiste em colocar todas as peças no tabuleiro sem que se tenha atingido o topo do tabuleiro.

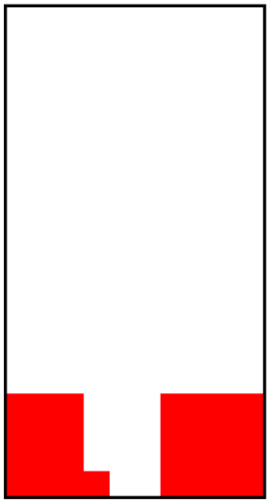
4.1.2 Testes Efetuados

Foram 13 os problemas escritos pelo nosso grupo para avaliar e comparar os dois algoritmos. O objetivo consistiu, como é obvio, em abranger todos os critérios descritos anteriormente de forma a fazer uma análise o mais completa e profunda possível.

Começamos pelo último critério enunciado na página anterior – estado do tabuleiro recebido. De seguida podemos ver as figuras que ilustram os 3 exemplos de tabuleiros usados nos testes.



Tabuleiro 1



Tabuleiro 2

O primeiro tabuleiro é um tabuleiro vazio, isto é, nenhuma célula encontra-se ocupada na altura em que damos o problema ao algoritmo para ser resolvido. Utilizamos este caso para verificar que ações toma o algoritmo quando ainda nenhuma peça foi colocada e, podemos mesmo dizer, não existe nenhuma posição favorável para obter pontos inicialmente. Este tabuleiro foi usado em 6 problemas com número incremental de peças por colocar (de 1 a 6).

O segundo tabuleiro vem do teste 25 do sistema de avaliação automático *Mooshak* em que já temos uma situação bem diferente de um tabuleiro vazio e onde já entram decisões condicionadas pelo estado do tabuleiro. Com isto queremos dizer que o objetivo deste tabuleiro era verificar qual a influência do facto de algumas células já estarem ocupadas no tabuleiro do estado inicial do problema. Desta maneira, conseguimos também testar com maior facilidade o critério relativo à qualidade das jogadas pois é possível alcançar um número elevado de pontos com as que temos disponíveis. Tal como o anterior, este tabuleiro foi usado em 6 problemas com número incremental de peças por colocar (de 1 a 6).

Esta foi a sequência de peças utilizada sem que tenha havido grande justificação para essa ordem ou mesmo para a natureza das peças. O grupo procurou diversificar nas peças de modo a que, sobretudo no Tabuleiro 2, fosse possível certo tipo de jogadas que permitissem alcançar um maior numero de pontos.

4.1.3 Resultados Obtidos

Consultar Anexo A1 e Anexo A2

4.1.4 Comparação dos Resultados Obtidos

- **Comentário ao Algoritmo de Procura em Profundidade**

Os resultados dos testes executados ao algoritmo de procura em profundidade são fáceis de analisar e de perceber a sua natureza. Começando pelos testes relativos ao tabuleiro vazio (Tabuleiro 1), reparamos que os tempos de execução não sobem dos 5 microssegundos e o número de estados explorados resume-se aos que separam o estado-inicial de um estado solução do problema. Estes números devem-se ao facto de não existir qualquer estudo/comparação de estados por parte deste algoritmo, isto é, entre a lista de estados por explorar escolhe-se o último estado a ser acrescentado à lista e somente isso (motivo pelo qual coloca todas a peças na coluna do tabuleiro mais à direita o possível). Desta maneira, o único processo a ocupar tempo no algoritmo é a expansão dos nós correspondentes a cada peça que é colocada (e que dá as ações possíveis a partir da ação anterior). Claro que isto tem as suas consequências, podemos observar que não conseguiu alcançar qualquer ponto mesmo com um número de peças para colocar que lhe permitiam para, pelo menos, ser capaz de 100 pontos ao remover uma linha.

Quanto aos testes relativos aos tabuleiros já preenchidos (Tabuleiro 2 e 3), pode-se dizer exatamente o mesmo que já foi dito para o tabuleiro vazio, podendo concluir que a eficiência deste algoritmo é completamente independente do estado do tabuleiro inicial. Esta dedução faz todo o sentido pois, como já dissemos, o algoritmo sabe sempre que o estado a explorar é o mais recente desta lista, não sofrendo as consequências das ações que toma e o que estas significam no estado do tabuleiro. Assim, avaliando o algoritmo sob o critério da qualidade das jogadas, podemos dizer que esta é mínima (ou melhor, inexistente), pois não há qualquer interesse em obter pontos, mas sim, em encontrar uma solução (que acaba por conseguir no mínimo de tempo, mas com as consequências observadas), outro dos critérios escolhidos.

- **Comentário ao Algoritmo de Procura A***

O algoritmo de procura A* tem muito mais que se diga, começando, desta vez, pelos testes com o Tabuleiro 2. Logo no caso em que temos apenas uma peça para colocar percebemos que, ao contrário do algoritmo anterior, este escolhe um estado da lista dos que tem para explorar de acordo com a heurística que recebe (que naturalmente, foi desenvolvida para procurar o melhor estado para obter o máximo de pontos). Isto porque coloca a peça de modo a obter pontos com a jogada (e não na coluna mais à direita do tabuleiro), neste caso 100. Outro aspeto importante de referir é a complexidade temporal e espacial deste algoritmo, pois à medida que subimos a dificuldade do problema (expandindo gradualmente a lista de pecas para colocar), é significativo o aumento do tempo de execução e da memória alocada para tal (associada ao número de estados explorados).

Quanto aos testes associados ao tabuleiro vazio, os resultados mostram claramente que o estado do tabuleiro inicial influência bastante a execução do algoritmo, com variações significativas no tempo e espaço do algoritmo. Como se pode observar nos resultados, com o tabuleiro sem qualquer célula ocupada, o algoritmo de procura A* demora cerca de 10 segundos (!!) e explora 7000 nós para resolver um problema com 4 peças a colocar, ao contrário dos 0,25 segundos e 195 nós para um problema com a mesma lista de peças, mas com o tabuleiro parcialmente preenchido. Esta diferença significativa de tempo e memória alocada deve-se ao processo de escolha do próximo estado a explorar por parte do algoritmo. O que acontece é que num tabuleiro preenchido é mais "fácil" diferenciar entre uma boa e uma má ação (com isto queremos dizer entre uma ação que nos permite ganhar pontos ou outra que nem tanto) porque temos possibilidades de ganhar pontos com jogadas que são, geralmente, escolhidas automaticamente pelo algoritmo para explorar (Atenção: nem sempre a ação que dá mais pontos é a que garante mais pontos no final do problema, isto é, existem ações que não satisfazem o máximo de pontos que seria possível ganhar nessa jogada mas permitem que uma próxima consiga ainda mais pontos do que seria possível escolhendo outra qualquer). Assim, como num tabuleiro vazio são inúmeros os estados resultantes que têm o mesmo valor segundo a heurística, o número de nós explorados vai ser maior (bem como o número de nós descartados).

É com o tabuleiro vazio e 6 peças para colocar que encontramos o primeiro erro grave neste algoritmo de procura. Isto porque não é encontrada qualquer solução (!!), ou seja, a função não encontra nenhum estado em que o tabuleiro não tenha o topo preenchido e todas as pecas da lista tenham sido colocadas. A razão para que isso aconteça é precisamente (e ironicamente) por causa de uma das medidas de otimização implementadas pelo grupo ao impor um limite no valor da soma da heurística com o custo-oportunidade (2500 foi o valor final), impedindo assim o algoritmo de explorar todos os nós que seriam naturalmente possíveis e sendo assim, depois de explorar toda a lista de estados por explorar (que não corresponde à verdadeira lista por causa do limite imposto), não encontra um estado solução porque estes (ou qualquer um que estivesse no seu caminho) tem um valor maior que 2500. A justificação deste valor deve-se sobretudo à função de custo-oportunidade que, à sexta peça, considera que já se perdeu muitos pontos (em relação à pontuação máxima possível com as peças por colocar – ver definição da função se necessário) e, por isso, atribui um valor bastante elevado. Isto foi claramente uma falha no projeto da nossa parte porque uma outra função de custo ou a não implementação do limite de 2500 corrigiria este problema. Sendo assim, este algoritmo da nossa autoria não pode ser considerado completo porque não consegue resolver o problema recebido.

- **Comparação dos Algoritmos**

Finalmente, comparemos os resultados de cada algoritmo e as diferenças são obvias... Em termos de complexidade temporal e espacial, a procura em profundidade "vence" por uma grande margem, demorando milésimas de segundo a resolver problemas que a procura-A* demora segundos e, a partir de determinado número de peças a colocar, minutos ou mesmo horas... Esta diferença de tempo de execução está ligada ao número de nós explorados, que no primeiro algoritmo correspondem praticamente ao número de peças a colocar (mais o estado inicial) (Atenção: isto verifica-se nestes testes, mas pode acontecer o número de estados explorar ser superior por não encontrar solução "à primeira") enquanto que no segundo chegamos a ter nós explorados na ordem dos milhares para os problemas mais difíceis. Por sua vez a diferença no número de estados explorados advêm da própria definição do algoritmo de profundidade que diz que o estado a ser explorado deve ser sempre o mais recente na lista de nós a explorar (poupando qualquer cálculos de heurísticas e funções a correr toda

uma lista com as consequências temporais e espaciais associadas) ao contrário do algoritmo de procura-A* que já envolve todo esse processo de cálculo de valores de heurística para cada estado para decidir em cada iteração, qual deles tem o valor mínimo, para de seguida explorá-lo e repetir o processo.

Infelizmente, na medida em que o projeto proposto era criar um jogador que obtivesse o máximo de pontos possível e não apenas chegar a uma solução dos problemas recebidos, a procura em profundidade não atribui qualquer importância à pontuação obtida no final (como podemos observar nos testes, não foram conseguidos quaisquer pontos com este algoritmo) e por isso, como já referimos, a qualidade das jogadas é considerada mínima ou mesmo inexistente. Por outro lado, o algoritmo de procura A* tem um desempenho bem mais elevado porque, apesar de não obter a pontuação máxima possível com o tabuleiro e peças para colocar em questão, alcança um número de pontos satisfatório, na opinião do grupo.

Concluindo, a procura em profundidade apresenta inúmeras vantagens: melhor complexidade temporal e especial, encontra sempre solução, apresentando uma execução independente do estado do tabuleiro recebido com o problema. Por outro lado, falha no critério mais importante e o mais decisivo para a "sentença final" – a qualidade das jogadas. Neste campo, a procura-A* é bastante optando por jogadas que garantem uma pontuação digna de um jogador de *Tetris* aceitável.

4.2 Estudo funções de custo/heurísticas

4.2.1 Critérios a analisar

Antes de avançar para os critérios é importante referir que esta foi uma área do projeto pouco trabalhada pelo nosso grupo e que acaba por explicar o algoritmo resultante bem como o seu desempenho longe do que gostávamos de apresentar. Desta maneira, o pouco estudo de funções de custos e heurísticas por nossa parte, resultou numa secção algo pobre de ideias.

Todos os testes foram realizados sobre o algoritmo de procura-A*, passando-lhe a heurística pretendida, através da *heurística-best* que consiste numa função em que aplica as várias heurísticas criadas, multiplicando-as por certos coeficientes e calculando a sua soma total. São estes coeficientes que procurámos otimizar através dos testes realizados.

Não houve qualquer proposta da nossa parte para uma função de custo que substituísse a já implementada da primeira parte *custo-oportunidade*, logo esta manteve-se como o custo-caminho.

O único critério que usámos para comparar e distinguir heurísticas foi a pontuação final obtida pelo algoritmo de procura-A*. Isto porque apenas nos interessámos em chegar à heurística que nos permitisse escolher as melhores jogadas, colocando as peças nas posições que, no final, permitissem o maior acumular de pontos.

4.2.2 Testes Efetuados

Depois de ter desenvolvido as heurísticas, tínhamos várias dúvidas em como iriamos usá-las no nosso problema. O teste desenvolvido para testarmos as várias tentativas que a função de *heurística-best* passou foi, de novo o Tabuleiro 3 descrito na seção de testes dos algoritmos, mas desta vez com uma lista de peças a colocar diferente. Desta maneira, temos 4 peças para colocar na ordem que mostramos a seguir:

A ideia por detrás da criação deste teste foi criar um tabuleiro simples, com um número de peças a colocar também não muito elevado, mas gerando uma situação possível de conseguir 800 pontos. Desta maneira, o nosso objetivo passou por conseguir chegar a esse valor, ou um valor mais perto possível deste. A verdade é que este teste é fácil de analisar a pontuação obtida, critério que avaliamos neste estudo.

4.2.3 Resultados Obtidos

Consultar Anexo B

4.2.4 Comparação dos Resultados Obtidos

Apesar dos resultados a nível de pontuação obtida falarem por si, comentemos os resultados obtidos para as diferentes variantes da função *heurística-best* criada pelo grupo. Curiosamente, o melhor resultado (600 pontos) foi obtido, eliminando as heurísticas associadas com o *Total Slopes* e os *Espaços Preenchidos*, baseando-se então apenas nas heurísticas da *Qualidade*, *Higher Slope* e *Buracos*. A verdade é que desta maneira já temos um valor que nos permite ganhar um número de pontos satisfatório, representando os estados com um valor que está perto da ideia que tínhamos para a nossa função de heurística.

4.3 Escolha da *procura-best*

A primeira decisão quanto à procura-best a desenvolver pelo grupo era escolher qual o algoritmo de procura que iriamos usar. As hipóteses eram muitas, sendo que tínhamos a procura em profundidade primeiro e a procura-A* já implementadas no nosso projeto, mas também possuíamos conhecimento de todo um outro leque de algoritmos vistos durante as aulas teóricas da cadeira.

O algoritmo em profundidade foi descartado quase de imediato porque o nosso objetivo não se tratava de encontrar uma mera solução do problema, mas sim encontrar uma boa solução (isto é, com o maior número de pontos alcançado). Como a procura-pp que escrevemos nem recebia heurísticas, apenas olhando para a profundidade da arvore de estados e se já encontrou solução, começámos a olhar para a procura-A*.

Este último era uma hipótese muito mais viável apesar de todos os problemas que encontrámos em otimizá-lo em tempo e em em espaço. Talvez agora, refletimos um pouco sobre esta escolha e percebemos e que talvez haveria algumas variantes deste algoritmo de procura que poderíamos ter usado e teria sido mais eficiente (talvez uma procura-A* iterativa em vez de limitada). Desta maneira, apenas foi necessário desenvolver uma função de heurística (talvez houvesse também a necessidade de chegar a uma nova função de custo) para passar como argumento.

Por fim, é aqui que entram os estudos das heurísticas porque acabámos a escolher a função de heurística que nos deu o melhor resultado no teste referido anteriormente: uma soma das heurísticas de *Buracos*, *Qualidade* e *Higher Slope* com coeficientes 500, 1 e 10 respetivamente.

5. Auto-Crítica

Apesar de não estar contemplado no template do relatório fornecido pelo corpo docente da disciplina, o nosso grupo achou por bem acrescer este capítulo dado o estado do projeto entregue e aos vários erros cometidos no seu desenvolvimento.

Queremos deixar claro que, acreditando que a avaliação da segunda e a terceira entrega do projeto será realizada de forma separada e independente, o esforço do nosso grupo foi redobrado na realização deste relatório. O objetivo foi não só compensar algumas das falhas apresentadas na parte prática como também percebermos o que falhou, o motivo de ter falhado e qual a possível solução para os resultados apresentados.

Em primeiro lugar, começamos por reforçar a ideia de que a otimização do algoritmo de procura-A* não foi otimizado da melhor maneira, o que acabou por se refletir em tudo o que se seguiu, incluindo, a execução deste algoritmo. O facto de impormos um limite de 2500 ao valor de um estado corta completamente o caminho que o algoritmo analisa na procura de uma solução para o problema, como seria de esperar. Esta medida aumenta de gravidade se olharmos para o algoritmo de *procura-best* e a *heurística-best* em que podemos perceber que este limite imposto impede a resolução de problemas, à partida, triviais. Tabuleiros iniciais com buracos já por preencher são um dos exemplos em que o nosso algoritmo falha redondamente, nem que seja apenas para colocar uma peça. Isto porque basta que o primeiro valor associado ao estado-inicial seja maior que 2500, para a procura terminar por aí sem encontrar solução.

Mais, a elaboração de heurísticas bem como a implementação e os testes de comparação não foram completos o suficiente para se obter um bom resultado com que ficássemos satisfeitos e com certeza que estávamos no caminho certo. Entretanto, o acumular de falhas no algoritmo parecia provocar ainda mais erros. A solução passaria então por otimizar o algoritmo de procura-A*, permitindo-nos testar de melhor forma outras soluções de heurísticas e funções de custo (não apresentámos nenhuma solução para este problema se bem que não era prioridade a nosso ver).

Outro problema que surgiu entretanto foi a ocorrência de *"stack overflows"* que advém do uso da função, de modo recursivo, *remove-nth*. Nunca é tarde demais, e percebemos agora que uma função recursiva não era uma boa solução para resolver este tipo de problemas e por isso apresentamos em anexo **(Anexo C)** uma solução (não recursiva) para a função *remove-nth* e que usámos em alguns testes apresentados anteriormente quando foi necessário (estas situações estão devidamente sinalizadas).

Concluindo, podíamos ter feito mais e melhor e, como tal, obtivemos um resultado abaixo do que estávamos à espera de conseguir. Por outro lado, a fase posterior à segunda entrega do projeto foi muito produtiva da nossa parte, trabalhando para a entrega de um relatório o mais completo possível face aos acontecimentos que o antecederam, escrevendo não só o que correu bem e está implementado corretamente, mas sobretudo apontando todas as falhas cometidas e que podiam ter sido corrigidas dentro do seu tempo. Acreditamos que este relatório acabou por nos ajudar bastante não só no projeto como nos conteúdos teóricos da cadeira porque, os aspetos que não trabalhámos sobre manipulação de algoritmos e criação e um jogador inteligente, acabámos por estudar enquanto escrevíamos estas mesmas palavras e isso foi bastante benéfico para o nosso grupo.

Anexo A1 – Testes ao Algoritmo de Procura em Profundidade Primeiro

Nº Teste	Descrição	Tempo de Execução	Estados Explorados	Pontuação Obtida
1	Tabuleiro vazio com 1 peça para colocar.	0.000891 s	2	0
2	Tabuleiro vazio com 2 peças para colocar.	0.001774 s	3	0
3	Tabuleiro vazio com 3 peças para colocar.	0.002911 s	4	0
4	Tabuleiro vazio com 4 peças para colocar.	0.003590 s	5	0
5	Tabuleiro vazio com 5 peças para colocar.	0.004227 s	6	0
6	Tabuleiro vazio com 6 peças para colocar.	0.004798 s	7	0
7	Tabuleiro parcialmente preenchido com 1 peça para colocar.	0.000836 s	2	0
8	Tabuleiro parcialmente preenchido com 2 peças para colocar.	0.002168 s	3	0
9	Tabuleiro parcialmente preenchido com 3 peças para colocar.	0.003138 s	4	0
10	Tabuleiro parcialmente preenchido com 4 peças para colocar.	0.003546 s	5	0
11	Tabuleiro parcialmente preenchido com 5 peças para colocar.	0.004198 s	6	0
12	Tabuleiro parcialmente preenchido com 6 peças para colocar.	0.004649 s	7	0

Notas:

O tempo de execução que consta nesta tabela e na tabela seguinte consiste na média aritmética de 10 execuções do teste (número de execuções considerada necessária dada a pouca variação de valores). A ordem das peças a colocar é a referida no relatório (t,l,j,i,l,i). O tabuleiro vazio refere-se ao Tabuleiro 1, enquanto que o tabuleiro parcialmente preenchido refere-se ao Tabuleiro 2 (pág. 11).

Anexo A2 – Testes ao Algoritmo de Procura-A*

Nº Teste	Descrição	Tempo de Execução	Estados Explorados	Pontuação Obtida
1	Tabuleiro vazio com 1 peça para colocar.	0.002248 s	2	0
2	Tabuleiro vazio com 2 peças para colocar.	0.022539 s	18	0
3	Tabuleiro vazio com 3 peças para colocar.	0.377156 s	203	0
4	Tabuleiro vazio com 4 peças para colocar.	9.985791 s	6970	100
5	Tabuleiro vazio com 5 peças para colocar.	24.637262 s	21483	100
6	Tabuleiro vazio com 6 peças para colocar.	26.642118 s *	23717	100
7	Tabuleiro parcialmente preenchido com 1 peça para colocar.	0.001849 s	2	100
8	Tabuleiro parcialmente preenchido com 2 peças para colocar.	0.013967 s	12	200
9	Tabuleiro parcialmente preenchido com 3 peças para colocar.	0.029833 s	23	400
10	Tabuleiro parcialmente preenchido com 4 peças para colocar.	0.249397 s	194	500
11	Tabuleiro parcialmente preenchido com 5 peças para colocar.	2.969672 s	1184	600
12	Tabuleiro parcialmente preenchido com 6 peças para colocar.	ERRO STACK OVERFLOW (85.647710 s) **	(11853)	(700)

* - O algoritmo terminou, mas não devolveu solução!

** - Erro devido à natureza recursiva da função *remove-nth,* apresentamos entre parêntesis o valor obtido com a correção da função.

Anexo B – Testes às Heurísticas

Nº Teste	Buracos (a)	Qualidade (b)	Higher Slope (c)	Total Slopes (d)	Espaços Preenchidos (e)	Pontuação Obtida
1	500	1	10	0	0	600
2	500	1	10	10	25	500
3	300	1	1	15	1	500
4	500	5	10	1	1	400
5	500	1	10	1	10	400
6	300	1	10	200	1	300

$$Heuristica\ Best = (a \times Buracos) + (b \times Qualidade) + (c \times Higher\ Slope) + (d \times Total\ Slopes) + (e \times Espaços\ Preenchidos)$$

Notas:

Todas as heurísticas foram avaliadas através do algoritmo de procura-A*, num tabuleiro parcialmente preenchido e com as peças para colocar já referidas no relatório (t,l,l,i). Os números presentes nas colunas associadas às heurísticas representam os coeficientes das heurísticas na função *heurística-best*.

Anexo C – Solução para o Problema de *Stack Overflow*

Reescrita da função *remove-nth* da seguinte maneira (de forma não recursiva):

```
(defun remove-nth (n lista)

  (let ((lista-res '())
        (lista-aux '()))


    (setf lista-res (nthcdr (+ n 1) lista))


    (setf lista-aux (reverse lista))


    (setf lista-aux (reverse (nthcdr (- (list-length
   lista-aux) n) lista-aux)))


    (setf lista-res (nconc lista-aux lista-res))


    lista-res


))
```