TÉCNICO LISBOA

Inteligência Artificial

1.º Semestre 2015/2016

Enunciado Projecto – *IA Tetris*

1 Introdução

Neste projecto pretende-se implementar um algoritmo de procura capaz de jogar uma variante do jogo *Tetris*, em que as peças por colocar são conhecidas à partida. O objectivo é tentar maximizar o número de pontos com as peças definidas. Para simplificar o jogo, só é permitido ao jogador escolher a rotação e a posição onde a peça irá cair. Uma vez escolhida a rotação e a posição, a peça deverá cair a direito sem qualquer movimento adicional.

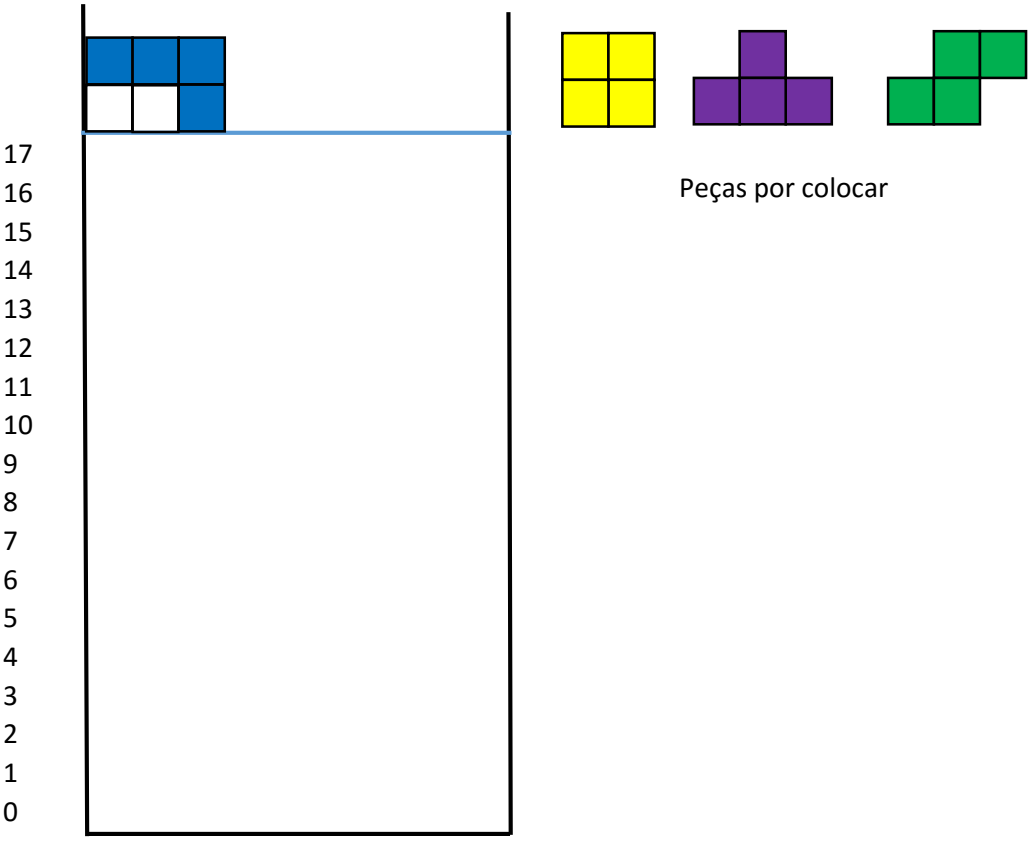

Figura 1Tabuleiro do jogo Tetris, ilustrando a próxima jogada escolhida, e as restantes peças por colocar

O tabuleiro do jogo é constituído por 10 colunas e 18 linhas. As linhas estão numeradas de 0 a 17, e as colunas de 0 a 9. Cada posição do tabuleiro pode estar vazia ou ocupada. Assim que uma peça é colocada sobre o tabuleiro, se existir alguma posição ocupada na linha 17 o jogo termina. Caso

contrário, todas as linhas que estejam completas[1] são removidas e o jogador ganha o seguinte número de pontos em função do número de linhas removidas:

Linhas	Pontos
1	100
2	300
3	500
4	800

Quando uma linha é removida, as linhas superiores a essa linha deverão descer como se vê na figura seguinte.

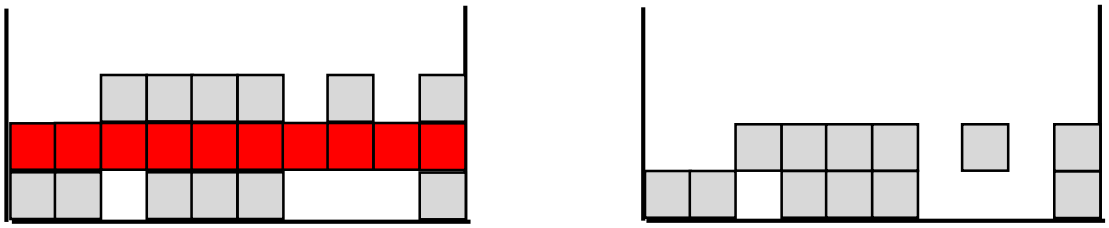

Figura 2 – tabuleiro resultante de se remover a linha do meio a vermelho.

2 Trabalho a realizar

O objectivo do projecto é escrever um programa em *Common Lisp*, que utilize técnicas de procura sistemática, para determinar a sequência de acções de modo a conseguir colocar todas as peças e tentando maximizar o número de pontos obtidos.

Para tal deverão realizar várias tarefas que vão desde a implementação dos tipos de dados usados na representação do tabuleiro, até à implementação dos algoritmos de procura e de heurísticas para guiar os algoritmos. Uma vez implementados os algoritmo de procura, os alunos deverão fazer um estudo/análise comparativa das várias versões dos algoritmos, bem como das funções heurísticas implementadas.

N**ão é necessário testarem os argumentos recebidos** pelas funções, a não ser que seja indicado explicitamente para o fazerem. Caso não seja dito nada, podem assumir que os argumentos recebidos por uma função estão sempre correctos.

2.1 Tipos Abstractos de Informação

2.1.1 Tipo Acção

O tipo Acção é utilizado para representar uma acção do jogador. Uma acção é implementada como um par cujo elemento esquerdo é um número de coluna que indica a coluna mais à esquerda escolhida para a peça cair, e cujo elemento direito corresponde a um *array* bidimensional com a configuração

[1] Uma linha é completa quando todas as posições dessa linha estão ocupadas.

geométrica da peça depois de rodada[2]. A figura seguinte mostra o exemplo de uma acção. O ficheiro *utils.lisp* define todas as configurações geométricas possíveis para cada peça[3].

```
(3 . #2A((T T nil)(nil T T)))
```

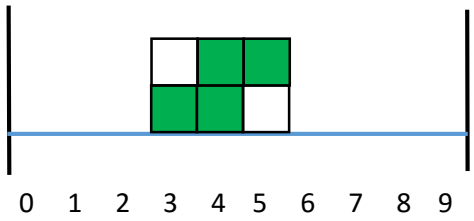

Figura 3- Representação em Common Lisp e representação visual para a acção de colocar a peça S com a orientação inicial na coluna 3

- cria-accao: inteiro x array → accao

  Este construtor recebe um inteiro correspondente à posição da coluna mais à esquerda a partir da qual a peça vai ser colocada, e um array com a configuração da peça a colocar, e devolve uma nova acção.

- accao-coluna: accao → inteiro

  Este selector devolve um inteiro correspondente à coluna mais à esquerda a partir da qual a peça vai ser colocada.

- accao-peca: accao → array

  Este selector devolve o array com a configuração geométrica exacta com que vai ser colocada.

2.1.2 Tipo tabuleiro

O tipo Tabuleiro é utilizado para representar o tabuleiro do jogo de *Tetris* com 18 linhas e 10 colunas, em que cada posição do tabuleiro pode estar preenchida ou não. Cabe aos alunos a escolha da representação mais adequada para este tipo.

- cria-tabuleiro: {}→ tabuleiro

  Este construtor não recebe qualquer argumento, e devolve um novo tabuleiro vazio.

- copia-tabuleiro: tabuleiro→ tabuleiro

  Este construtor recebe um tabuleiro, e devolve um novo tabuleiro com o mesmo conteúdo do tabuleiro recebido. O tabuleiro devolvido deve ser um objecto computacional diferente e deverá garantir que qualquer alteração feita ao tabuleiro original não deve ser repercutida no novo tabuleiro e vice-versa.

- tabuleiro-preenchido-p: tabuleiro x inteiro x inteiro → lógico

---

[2] Um *array* em que cada posição pode ter o valor *T* caso essa posição esteja ocupada por uma parte da peça e *nil* caso contrário. A posição inicial do *array (0,0)* corresponde sempre à parte da peça mais à esquerda e mais abaixo.
[3] Guardadas nas constantes peca-i0, peca-i1, peca-j0,etc.

Este selector recebe um tabuleiro, um inteiro correspondente ao número da linha e um inteiro correspondente ao número da coluna, e devolve o valor lógico verdade se essa posição estiver preenchida, e falso caso contrário.

- tabuleiro-altura-coluna: tabuleiro x inteiro → inteiro

  Este selector recebe um tabuleiro, um inteiro correspondente ao número de uma coluna, e devolve a altura de uma coluna, ou seja a posição mais alta que esteja preenchida dessa coluna. Uma coluna que não esteja preenchida deverá ter altura 0.

- tabuleiro-linha-completa-p: tabuleiro x inteiro → lógico

  Este reconhecedor recebe um tabuleiro, um inteiro correspondente ao número de uma linha, e devolve o valor lógico verdade se todas as posições da linha recebida estiverem preenchidas, e falso caso contrário.

- tabuleiro-preenche!: tabuleiro x inteiro x inteiro → {}

  Este modificador recebe um tabuleiro, um inteiro correspondente ao número linha e um inteiro correspondente ao número da coluna, e altera o tabuleiro recebido para na posição correspondente à linha e coluna passar a estar preenchido. Se o número de linha e de coluna recebidos não forem válidos (i.e. não estiverem entre 0 e 17, e 0 e 9), esta função não deverá fazer nada. O valor devolvido por desta função não está definido[4].

- tabuleiro-remove-linha!: tabuleiro x inteiro → {}

  Este modificador recebe um tabuleiro, um inteiro correspondente ao número de linha, e altera o tabuleiro recebido removendo essa linha do tabuleiro, e fazendo com que as linhas por cima da linha removida desçam uma linha. As linhas que estão por baixo da linha removida não podem ser alteradas. O valor devolvido por desta função não está definido.

- tabuleiro-topo-preenchido-p: tabuleiro → lógico

  Este reconhecedor recebe um tabuleiro, e devolve o valor lógico verdade se existir alguma posição na linha do topo do tabuleiro (linha 17) que esteja preenchida, e falso caso contrário.

- tabuleiros-iguais-p: tabuleiro x tabuleiro → lógico

  Este teste recebe dois tabuleiros, e devolve o valor lógico verdade se os dois tabuleiros forem iguais (i.e. tiverem o mesmo conteúdo), e falso caso contrário.

- tabuleiro->array: tabuleiro → array

  Este transformador de saída recebe um tabuleiro e devolve um novo array com 18 linhas e 10 colunas, que para cada linha e coluna deverá conter o valor lógico correspondente a cada posição do tabuleiro. O array retornado deverá garantir que qualquer alteração feita ao tabuleiro original não deve ser repercutida no novo array e vice-versa.

[4] Ou seja, não interessa o que é devolvido, mas também não podem usar o valor devolvido quando invocam a função.

- array->tabuleiro: array → tabuleiro

  Este transformador de entrada recebe um array com 18 linhas e 10 colunas cujas posições têm o valor lógico T ou Nil, e constrói um novo tabuleiro com o conteúdo do array recebido. O tabuleiro devolvido deverá garantir que qualquer alteração feita ao array original não deve ser repercutida no novo tabuleiro e vice-versa.

2.1.3 Tipo Estado

O tipo estado representa o estado de um jogo de *Tetris*. Este tipo deverá ser implementado **obrigatoriamente** como uma **estrutura**[5] em *Common Lisp* com os seguintes campos:

- pontos – o número de pontos conseguidos até ao momento no jogo;
- pecas-por-colocar – uma lista com as peças que ainda estão por colocar, pela ordem de colocação. As peças nesta lista são representadas pelo símbolo correspondente à letra da peça, i.e. i,j,l,o,s,z,t[6];
- pecas-colocadas – uma lista com as peças já colocadas no tabuleiro (representadas também pelo símbolo). Esta lista deve encontrar-se ordenada da peça mais recente para a mais antiga;
- Tabuleiro – o tabuleiro com as posições actualmente preenchidas do jogo.

Para além das operações construídas automaticamente para estruturas (ex: selectores e modificadores), deverá implementar os seguintes operadores adicionais sobre estados.

- copia-estado: estado → estado

  Este construtor recebe um estado e devolve um novo estado cujo conteúdo deve ser copiado a partir do estado original. O estado devolvido deverá garantir que qualquer alteração feita ao estado original não deve ser repercutida no novo estado e vice-versa.

- estados-iguais-p: estado x estado → lógico

  Este teste recebe dois estados, e devolve o valor lógico verdade se os dois estados forem iguais (i.e. tiverem o mesmo conteúdo) e falso caso contrário.

- estado-final-p: estado → lógico

  Este reconhecedor recebe um estado e devolve o valor lógico verdade se corresponder a um estado final onde o jogador já não possa fazer mais jogadas e falso caso contrário. Um estado é considerado final se o tabuleiro tiver atingido o topo ou se já não existem peças por colocar.

2.1.4 Tipo problema

O tipo problema representa um problema genérico de procura. Este tipo deverá ser implementado **obrigatoriamente** como uma **estrutura** em *Common Lisp* com os seguintes campos:

- estado-inicial – contem o estado inicial do problema de procura;

[5] No inglês ***structure***.
[6] Ver ficheiro *utils.lisp*.

- solucao – função que recebe um estado e devolve T se o estado for uma solução para o problema de procura, e nil caso contrário;
- accoes – função que recebe um estado e devolve uma lista com todas as acções que são possíveis fazer nesse estado;
- resultado – função que dado um estado e uma acção devolve o estado sucessor que resulta de executar a acção recebida no estado recebido;
- custo-caminho – função que dado um estado devolve o custo do caminho desde o estado inicial até esse estado.

2.2 Funções a implementar

Para além dos tipos de dados especificados na secção anterior, é obrigatória também a implementação das seguintes funções.

2.2.1 Funções do problema de procura

Estas funções podem ser usadas como funções do problema de procura, que depois serão usadas pelos algoritmos de procura:

- solucao: estado → lógico

  Esta função recebe um estado, e devolve o valor lógico verdade se o estado recebido corresponder a uma solução, e falso contrário. Um estado do jogo *Tetris* é considerado solução se o topo do tabuleiro não estiver preenchido e se já não existem peças por colocar, ou seja, todas as peças foram colocadas com sucesso (independentemente de terem ou não sido obtidos pontos).

- accoes: estado → lista de acções

  Esta função recebe um estado e devolve uma lista de acções correspondendo a todas as acções válidas que podem ser feitas com a próxima peça a ser colocada. Uma acção é considerada válida mesmo que faça o jogador perder o jogo (i.e. preencher a linha do topo). Uma acção é inválida se não for fisicamente possível dentro dos limites laterais do jogo[7][8]. Por exemplo colocar a peça i deitada na última coluna do tabuleiro, ou tentar colocar a peça s com a orientação inicial na coluna 8[9], tal como se pode ver na imagem seguinte.

Figura 4- Exemplo de acção inválida

  A ordem com que são devolvidas as acções na lista é muito importante. À frente da lista devem estar obrigatoriamente as acções correspondentes à orientação inicial da peça, percorrendo

[7] Repare que é sempre possível colocar uma peça na coluna 0, pois a posição (0,0) de cada peça corresponde sempre à parte mais à esquerda e mais abaixo da mesma.
[8] Os limites verticais não tornam uma acção inválida. Ou seja, jogar uma peça que ocupe a linha 17 e continue para cima é valida, embora vá fazer com que o jogador perca o jogo no estado sucessor.
[9] Acção representada em *Common Lisp* como `(8 . #2A((T T nil)(nil T T)))`.

todas as colunas possíveis da esquerda para a direita. Depois é escolhida uma nova orientação, odando a peça 90º no sentido horário, e volta-se a gerar para todas as colunas possíveis da esquerda para a direita. No entanto, se ao rodar uma peça obter uma configuração geométrica já explorada anteriormente (como por exemplo no caso da peça O em que todas as rotações correspondem à mesma configuração) não devem ser geradas novamente as acções[10].

- resultado: estado x accao → estado

  Esta função recebe um estado e uma acção, e devolve um novo estado que resulta de aplicar a acção recebida no estado original. Atenção, o estado original não pode ser alterado em situação alguma. Esta função deve actualizar as listas de peças, colocar a peça especificada pela acção na posição correcta do tabuleiro. Depois de colocada a peça, é verificado se o topo do tabuleiro está preenchido. Se estiver, não se removem linhas e devolve-se o estado. Se não estiver, removem-se as linhas e calculam-se os pontos obtidos.

- qualidade: estado → inteiro

  Os algoritmos de procura informada estão concebidos para tentar minimizar o custo de caminho. No entanto, se quisermos maximizar os pontos obtidos, podemos olhar para isto como um problema de maximização de qualidade. Para podermos usar a qualidade com os algoritmos de procura melhor primeiro, uma solução simples é convertermos a qualidade num valor negativo de custo. Assim sendo, um estado com mais pontos irá ter um valor menor (negativo) e terá prioridade para o mecanismo de escolha do próximo nó a ser expandido.

  Portanto, a função qualidade recebe um estado e retorna um valor de qualidade que corresponde ao valor negativo dos pontos ganhos até ao momento.

- custo-oportunidade: estado → inteiro

  Uma representação alternativa para um problema de maximização de qualidade, é considerar que por cada acção podemos potencialmente ganhar um determinado valor, e que o custo é dado pelo facto de não conseguirmos ter aproveitado ao máximo a oportunidade. Assim sendo o custo de oportunidade pode ser calculado como a diferença entre o máximo possível e o efetivamente conseguido. Portanto esta função, dado um estado, devolve o custo de oportunidade de todas as acções realizadas até ao momento, assumindo que é sempre possível fazer o máximo de pontos por cada peça colocada[11]. Ao usarmos esta função como custo, os algoritmos de procura irão tentar minimizar o custo de oportunidade.

2.2.2 Procuras

As funções descritas nesta subsecção correspondem aos algoritmos de procura a implementar para determinar a sequência de acções de modo a colocar todas as peças.

- procura-pp: problema → lista acções

[10] As constantes definidas no ficheiro *utils.lisp* já estão ordenadas por esta ordem de rotação, pelo que basta por exemplo no caso da peça t gerarem as colunas possíveis para as peças:peca-t0,peca-t1,peca-t2,peca-t3.

[11] Tendo em conta as simplificações usadas no jogo, a pontuação máxima por cada peça é dada por: i – 800; j – 500, l – 500,s – 300, z – 300, t – 300, o – 300.

Esta função recebe um problema e usa a procura em profundidade primeiro em árvore para obter uma solução para resolver o problema. Devolve uma lista de acções que se executada pela ordem especificada irá levar do estado inicial a um estado objectivo. Deve ser utilizado um critério de *Last In First Out*, pelo que o último nó a ser colocado na fronteira deverá ser o primeiro a ser explorado a seguir. Devem também ter o cuidado do algoritmo ser independente do problema, ou seja deverá funcionar para este problema do *Tetris*, mas deverá funcionar também para qualquer outro problema[12] de procura.

- procura-A*: problema x heurística → lista acções

  Esta função recebe um problema e uma função heurística, e utiliza o algoritmo de procura *A** em árvore para tentar determinar qual a sequência de acções de modo a maximizar os pontos obtidos. A função heurística corresponde a uma função que recebe um estado e devolve um número, que representa a estimativa do custo/qualidade[13] a partir desse estado até ao melhor estado objectivo. Em caso de empate entre dois nós com igual valor de *f* na lista deve ser escolhido o último a ter sido colocado na fronteira. Devem também ter o cuidado do algoritmo ser independente do problema.

- procura-best: array x lista peças → lista acções

  Esta função recebe um *array* correspondente a um tabuleiro e uma lista de peças por colocar, inicializa o estado e a estrutura problema com as funções escolhidas pelo grupo, e irá usar a melhor procura e a melhor heurística e melhor função custo/qualidade feita pelo grupo para obter a sequência de acções de modo a conseguir colocar todas as peças no tabuleiro com o máximo de pontuação. No entanto, tenham em consideração que esta função irá ter um limite de tempo para retornar um resultado, portanto não vos serve de nada retornar a solução óptima se excederem o tempo especificado[14]. É importante encontrar um compromisso entre a pontuação obtida e o tempo de execução do algoritmo. Esta função irá ser a função usada para avaliar a qualidade da vossa versão final. Se assim o entenderem, nesta função já podem usar implementações e optimizações específicas para o jogo do *Tetris*.

É importante ter em conta, que para além destas funções explicitamente definidas (que serão testadas automaticamente), **na 2.ª fase do projecto deverão implementar técnicas adicionais de optimização, várias heurísticas, ou até mesmo funções de custo alternativas, e testá-las. Cabe aos alunos decidir que técnicas/heurísticas adicionais irão precisar para que o vosso algoritmo final procura-best seja o melhor possível**.

2.3 Estudo e análise dos algoritmos e heurísticas implementadas

Na parte final do projecto, é obrigatório os alunos compararem as diferentes variantes das procuras, algoritmos e heurísticas usadas para resolver o jogo do *Tetris* especificado, e perceberem as diferenças entre elas. Para isso deverão medir vários factores relevantes, e comparar os algoritmos num conjunto significativo de exemplos. Deverão também tentar analisar/justificar o porquê dos resultados obtidos.

---

[12] Portanto, é desaconselhado o uso de variáveis globais e usar/aceder directamente a funções específicas do *Tetris*. Tudo o que precisarem vai estar dentro do tipo problema recebido.

[13] Repare que no caso de se usar qualidade, como esta vai corresponder a um valor negativo, o algoritmo A* vai continuar a selecionar o nó com menor valor de f.

[14] Por exemplo a procura de custo uniforme garante a solução óptima mas irá levar demasiado tempo.

Os resultados dos testes efectuados deverão ser usados para escolher as melhores procuras e as melhores funções de custo/heurísticas a serem usadas. Esta escolha deverá ser devidamente justificada no relatório do projecto.

Nesta fase final é também pretendido que os alunos escrevam um relatório sobre o projecto realizado. Para além dos testes efectuados e da análise correspondente, o relatório do projecto deverá conter também informação acerca da implementação de alguns tipos e funções. Por exemplo, qual a representação escolhida para cada tipo (e porquê), qual a função de utilidade utilizada, como foram implementas as procuras e que optimizações foram efectuadas, quais as funções heurísticas implementadas e como é que foram implementadas, etc. Será disponibilizado um *template* do relatório para que os alunos tenham uma ideia melhor do que é necessário incluir no relatório final do projecto.

3 Ficheiro utils

Foi fornecido um ficheiro de utilitários juntamente com o enunciado, que define a configuração geométrica para cada rotação possível de cada peça. Para além disso, fornece também um conjunto de funções para gerar tabuleiros aleatórios e listas de peças aleatórias. Mas mais importante, o ficheiro *utils* fornece uma função muito útil para visualizar a execução das jogadas devolvidas pelo vosso algoritmo de procura, e testar assim a qualidade das vossas funções.

A função executa-jogadas recebe um estado inicial e uma lista de acções (a lista devolvida por um algoritmo de procura), e vai desenhando o estado resultante de ir colocar as peças de acordo com as acções recebidas, mostrando os pontos obtidos. Para avançar entre ecrãs, deverá premir a tecla "*Enter*".

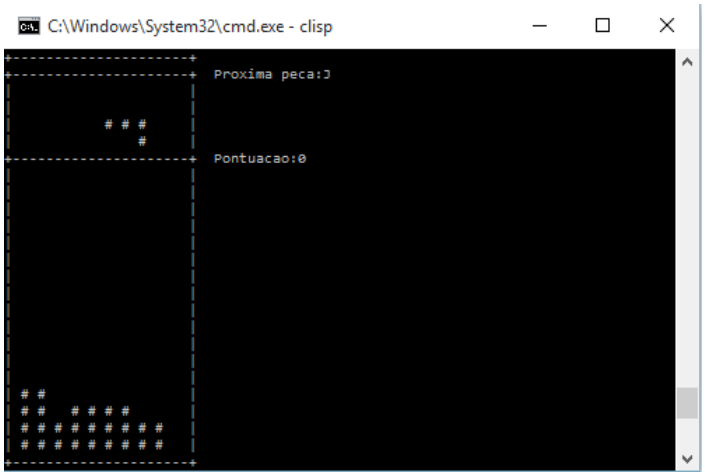

Figura 5- Exemplo de visualização usando a função executa-jogadas

4 Entregas, Prazos, Avaliação e Condições de Realização

4.1 Entregas e Prazos

A realização do projecto divide-se em 3 entregas. As duas primeiras entregas correspondem à implementação do código do projecto, enquanto que a 3.ª entrega corresponde à entrega do relatório do projecto.

4.1.1 1.ª Entrega

Na primeira entrega pretende-se que os alunos implementem os **tipos de dados** descritos no enunciado (**secção 2.1**) bem como as funções correspondentes à **secção 2.2.1**. A entrega desta primeira parte será feita por via electrónica através do sistema ***Mooshak***, até às **23:59** do dia **09/11/2014**. Depois desta hora, não serão aceites projectos sob pretexto algum[15].

Deverá ser submetido um ficheiro *.lisp* contendo o código do seu projecto. O ficheiro de código deve conter em comentário, na primeira linha, os números e os nomes dos alunos do grupo, bem como o número do grupo. Não é necessário incluir os ficheiros disponibilizados pelo corpo docente.

4.1.2 2.ª Entrega

Na segunda entrega do projecto os alunos devem implementar o resto das funcionalidades descritas no enunciado (ver **secção 2.2.2**), incluindo as várias variantes dos algoritmos de procura, e as funções heurísticas pedidas. Deverão também já nesta fase fazer os testes que permitirão escolher qual a melhor procura/heurística (embora não seja necessário incluir os testes no ficheiro de código submetido). A entrega da segunda parte será feita por via electrónica através do sistema ***Mooshak***, até às **23:59** do dia **30/11/2014**. Depois desta hora, não serão aceites projectos sob pretexto algum.

Deverá ser submetido um ficheiro *.lisp* contendo o código do seu projecto. O ficheiro de código deve conter em comentário, na primeira linha, os números e os nomes dos alunos do grupo, bem como o número do grupo. Não é necessário incluir os ficheiros disponibilizados pelo corpo docente.

4.1.3 3.ª Entrega

Na terceira entrega, os alunos deverão trabalhar exclusivamente no relatório do projecto. Não será aceite/avaliada qualquer nova entrega de código por parte dos alunos. Não existem excepções. O relatório do projecto deverá ser entregue (em .doc ou .pdf) exclusivamente por via electrónica no sistema ***FENIX***, até às **12:00** (meio dia) do dia **9/12/2014**.

4.2 Avaliação

As várias entregas têm pesos diferentes no cálculo da nota do Projecto. A 1.ª entrega (em código) corresponde a **30%** da nota final do projecto (ou seja 6 valores). A 2.ª entrega (em código) corresponde a **35%** da nota final do projecto (ou seja 7 valores). Finalmente a 3.ª entrega, referente ao relatório, corresponde aos restantes **35%** da nota final do projecto (ou seja 7 valores).

A avaliação da 1.ª entrega será feita exclusivamente com base na execução correcta (ou não) das funções pedidas.

A avaliação da 2.ª entrega terá duas componentes:

- A 1.ª componente vale 6 valores e corresponde à avaliação da correcta execução das funções pedidas e à qualidade do jogador de *Tetris*. A avaliação da qualidade do vosso algoritmo é feita usando-o para resolver uma série de tabuleiros de *Tetris* com dificuldade incremental, e com um montante de tempo limitado. Será tida em consideração a capacidade de retornar uma solução, bem como a quantidade de pontos obtida para cada tabuleiro. A 2.ª componente vale 1 valor e

[15] Note que o limite de 10 submissões simultâneas no sistema *Mooshak* implica que, caso haja um número elevado de tentativas de submissão sobre o prazo de entrega, alguns grupos poderão não conseguir submeter nessa altura e verem-se, por isso, impossibilitados de submeter o código dentro do prazo.

corresponde a uma avaliação manual da qualidade do código produzido. Serão analisados factores como comentários, facilidade de leitura (nomes e indentação), estilo de programação e utilização de abs. procedimental.

Finalmente, a avaliação da 3.ª entrega corresponde à avaliação do relatório entregue. No *template* de relatório que será disponibilizado mais tarde, poderão encontrar informação mais detalhada sobre esta avaliação.

4.3 Condições de realização

O código desenvolvido deve compilar em CLISP 2.49 sem qualquer "warning" ou erro. Todos os testes efectuados automaticamente, serão realizados com a versão compilada do vosso projecto. Aconselhamos também os alunos a compilarem o código para os seus testes de comparações entre algoritmos/heurísticas, pois a versão compilada é consideravelmente mais rápida que a versão não compilada a correr em *Common Lisp*.

No seu ficheiro de código não devem ser utilizados caracteres acentuados ou qualquer carácter que não pertença à tabela ASCII, sob pena de falhar todos os testes automáticos. Isto inclui comentários e cadeias de caracteres.

É prática comum a escrita de mensagens para o ecrã, quando se está a implementar e a testar o código. Isto é ainda mais importante quando se estão a testar/comparar os algoritmos. No entanto, **não se esqueçam de remover/comentar as mensagens escritas no ecrã na versão final** do código entregue. Se não o fizerem, correm o risco dos testes automáticos falharem, e irão ter uma má nota na execução.

A avaliação da execução do código do projecto será feita automaticamente através do sistema *Mooshak*, usando vários testes configurados no sistema. O tempo de execução de cada teste está limitado, bem como a memória utilizada. Só poderá efectuar uma nova submissão pelo menos 15 minutos depois da submissão anterior. Só são permitidas 10 submissões em simultâneo no sistema, pelo que uma submissão poderá ser recusada se este limite for excedido. Nesse caso tente mais tarde.

Os testes considerados para efeitos de avaliação podem incluir ou não os exemplos disponibilizados, além de um conjunto de testes adicionais. O facto de um projecto completar com sucesso os exemplos fornecidos não implica, pois, que esse projecto esteja totalmente correcto, pois o conjunto de exemplos fornecido não é exaustivo. É da responsabilidade de cada grupo garantir que o código produzido está correcto.

Duas semanas antes do prazo da 1.ª entrega (isto é, na Segunda-feira, 27 de Outubro), serão publicadas na página da cadeira as instruções necessárias para a submissão do código no *Mooshak*. Apenas a partir dessa altura será possível a submissão por via electrónica. Até ao prazo de entrega poderá efectuar o número de entregas que desejar, sendo utilizada para efeitos de avaliação a última entrega efectuada. Deverão portanto verificar cuidadosamente que a última entrega realizada corresponde à versão do projecto que pretendem que seja avaliada. Não serão abertas excepções.[16]

Pode ou não haver uma discussão oral do trabalho e/ou uma demonstração do funcionamento do programa (será decidido caso a caso).

[16] Note que se efectuar uma submissão no Mooshak a menos de 15 minutos do prazo de entrega, fica impossibilitado de efectuar qualquer outra submissão posterior.

Projectos muito semelhantes serão considerados cópia e rejeitados. A detecção de semelhanças entre projectos será realizada utilizando *software* especializado[17] e caberá exclusivamente ao corpo docente a decisão do que considera ou não cópia. Em caso de cópia, todos os alunos envolvidos terão 0 no projecto e serão reprovados na cadeira.

5 Competição do Projecto

Vai ser realizada uma competição entre todos os projectos submetidos para determinar quais os melhores projectos a resolver tabuleiros de *Tetris*. Para poderem participar na competição, a vossa implementação tem que passar todos os testes de execução referentes às funções e tipos de dados definidos neste enunciado. A função a ser usada na competição é a função *procura-best*.

Os 3 melhores classificados na competição, ou seja os projectos que consigam resolver os tabuleiros com melhor pontuação serão premiados com as seguintes bonificações:

- 1.º Lugar – 1,5 valores de bonificação na nota final do projecto
- 2.º Lugar – 1,0 valores de bonificação na nota final do projecto
- 3.º Lugar – 0,5 valores de bonificação na nota final do projecto

[17] Ver http://theory.stanford.edu/~aiken/moss/