TÉCNICO LISBOA

Inteligência Artificial

1.º Semestre 2013/2014

Enunciado Projecto – Moedas e Fios

1 Introdução

O jogo Moedas e Fios é uma variante do jogo Timberiche (ou Dots and Boxes[1] na terminologia anglosáxonica). Neste jogo, o tabuleiro inicial é constituído por um conjunto de moedas que estão espalhadas pelas várias posições do tabuleiro, e por um conjunto de fios que ligam pares de moedas adjacentes. Dois jogadores alternam turnos cortando os fios do tabuleiro. Quando um jogador elimina o último fio ligado a uma moeda, essa moeda é removida do jogo sendo contabilizada nos pontos do jogador, e adicionalmente o jogador joga de novo. O jogo termina quando todos os fios são cortados, e o jogador com maior valor de moedas acumulado vence o jogo. A Figura 1 mostra um exemplo de um tabuleiro do jogo Moedas e Fios.

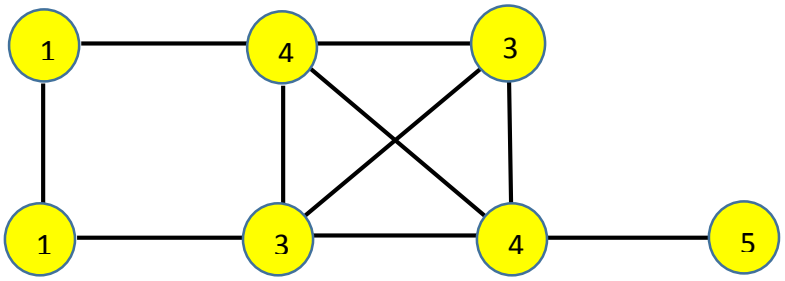

Figura 1 – exemplo de tabuleiro do jogo Moedas e Fios. Se o próximo jogador eliminar o fio ligado à moeda de valor 5, vai ganhar 5 pontos e jogar de novo.

2 Trabalho a realizar

O objectivo do projecto é escrever um programa em Lisp que implemente um jogador automático para jogar o jogo Moedas e Fios. Para tal deverão realizar várias tarefas que vão desde a implementação de alguns tipos de dados usados na representação do jogo e do tabuleiro, até à implementação da inteligência artificial do jogador automático usando uma ou mais variantes do algoritmo Minimax. Uma vez implementado o jogador automático, os alunos deverão fazer um estudo/análise comparativa das várias versões do algoritmo minimax, bem como das funções de avaliação/heurísticas implementadas.

[1] Para mais informação sobre a versão original do jogo consultem http://en.wikipedia.org/wiki/Dots_and_Boxes)

Não é necessária a implementação da interface para jogar o jogo Moedas e Fios. Esta função (bem como outras que podem ser utilizadas para testar o jogador automático) é fornecida pelo corpo docente da cadeira, e encontra-se no ficheiro interface-moedas.lisp.

Também **não é necessário testarem os argumentos recebidos** pelas funções. Podem assumir que os argumentos recebidos por uma função estão sempre correctos.

2.1 Tipos Abstractos de Informação

2.1.1 Tipo Posição

O tipo Posição é utilizado para representar uma posição (através da sua linha e coluna) do tabuleiro. As operações básicas associadas a este tipo são:

- cria-posicao: inteiro x inteiro → posição

  Este construtor recebe um inteiro (≥0) correspondente a uma linha, e outro inteiro (≥0) correspondente a uma coluna, e retorna uma posição com a linha e coluna recebidas.

- posicao-linha: posição → inteiro

  Este selector, dada uma posição p, retorna a linha (um inteiro ≥ 0) correspondente à posição.

- posicao-coluna: posição → inteiro

  Este selector, dada uma posição p, retorna a coluna (um inteiro ≥ 0) correspondente à posição.

- posicao-p: universal → lógico

  Este reconhecedor recebe um objecto x qualquer e retorna T se o objecto recebido for uma posição, e NIL caso contrário.

- posicoes-iguais-p: posição x posição → lógico

  Este teste recebe duas posições p1 e p2, e retorna T se p1 e p2 corresponderem à mesma posição (i.e. mesma linha e coluna), e NIL caso contrário.

2.1.2 Tipo Fio

O tipo fio é utilizado para representar um fio do tabuleiro que liga um par de moedas. As operações básicas associadas a este tipo são:

- cria-fio: inteiro x posição x posição → fio

  Este construtor recebe um inteiro (> 0) que corresponde ao identificador do fio no tabuleiro, uma posição que corresponde à posição da moeda de origem no tabuleiro, e outra posição que corresponde à posição da moeda destino no tabuleiro. Retorna o fio que liga as duas posições.

- fio-id: fio → inteiro

  Este selector recebe um fio e retorna o identificador desse fio no tabuleiro, que corresponde a um inteiro > 0.

- fio-origem: fio → posição

  Este selector recebe um fio e retorna a posição correspondente à origem do fio.

- fio-destino: fio → posição
  Este selector recebe um fio e retorna a posição correspondente ao destino do fio.
- fio-p: universal → lógico

  Este reconhecedor recebe um objecto qualquer x, e retorna T se x for um fio, e NIL caso contrário.

2.1.3 Tipo Tabuleiro

O tipo Tabuleiro é usado para representar o estado de um tabuleiro do jogo Moedas e Fios, contendo informação sobre as moedas e fios existentes no tabuleiro. As operações básicas associadas a este tipo são:

- cria-tabuleiro: inteiro x inteiro → tabuleiro

  Este construtor recebe um inteiro (≥ 2) correspondente ao número de linhas l, e um inteiro (≥ 2) correspondente ao número de colunas c. Retorna um tabuleiro vazio com l linhas e c colunas.

- copia-tabuleiro: tabuleiro → tabuleiro

  Este construtor recebe um tabuleiro, e cria um **novo** tabuleiro cujo conteúdo é o mesmo do tabuleiro recebido. O tabuleiro retornado tem que ser uma nova cópia, i.e. se o tabuleiro original for alterado, o tabuleiro retornado não pode mudar.

- tabuleiro-linhas: tabuleiro → inteiro

  Este selector recebe um tabuleiro e devolve o número de linhas do tabuleiro.

- tabuleiro-colunas: tabuleiro → inteiro

  Este selector recebe um tabuleiro e devolve o número de colunas do tabuleiro

- tabuleiro-fios: tabuleiro → lista de fios

  Este selector recebe um tabuleiro e devolve uma lista com todos os fios existentes no tabuleiro.

- tabuleiro-fio-com-id: tabuleiro x inteiro → fio

  Esse selector recebe um tabuleiro e um inteiro (>0) correspondente ao identificador de um fio, e retorna o fio do tabuleiro com esse identificador. Caso não exista é retornado NIL.

- tabuleiro-fios-posicao: tabuleiro x posição → lista de fios

  Este selector recebe um tabuleiro e uma posição do tabuleiro, e retorna uma lista com todos os fios que estão ligados à posição recebida (i.e. com origem ou destino na posição).

- tabuleiro-moeda-posicao: tabuleiro x posição → inteiro

  Este selector recebe um tabuleiro e uma posição do tabuleiro, e retorna o valor da moeda que está na posição recebida. Este valor é um inteiro entre 1 e 9. Caso não exista moeda na posição recebida é retornado NIL.

- tabuleiro-total-moedas: tabuleiro → inteiro

  Este selector recebe um tabuleiro e devolve a soma do valor de todas as moedas que estão no tabuleiro. Se não existir nenhuma moeda no tabuleiro, deverá retornar 0.

- tabuleiro-adiciona-fio!: tabuleiro x posição x posição → {}

  Este modificador recebe um tabuleiro, e duas posições adjacentes, e cria um fio que liga as duas posições recebidas, adicionando-o ao tabuleiro. Esta função não retorna nada, mas **altera** o tabuleiro recebido para que este contenha o novo fio criado. O identificador do fio criado corresponde à ordem de inserção do mesmo no tabuleiro, ou seja, o 1.º fio a ser adicionado deverá ter o identificador 1, o 2.º o identificador 2, etc. Não podem existir dois fios com o mesmo identificador no tabuleiro.

- tabuleiro-adiciona-moeda-posicao!: tabuleiro x posição x inteiro → {}

  Este modificador recebe um tabuleiro, uma posição do tabuleiro, e o valor de uma moeda (inteiro entre 1 e 9), e adiciona a moeda à posição recebida do tabuleiro. Se já existia uma moeda nessa posição, a moeda anterior é substituída. Esta função não retorna nada, mas **altera** o tabuleiro recebido de modo a que este contenha a moeda.

- tabuleiro-remove-fio-com-id!: tabuleiro x inteiro → {}

  Este modificador recebe um tabuleiro, e um inteiro correspondente ao identificador de um fio do tabuleiro a remover. Esta função não retorna nada, mas **altera** o tabuleiro recebido removendo o fio em questão. Caso não exista no tabuleiro nenhum fio com o identificador recebido, nada é removido e o tabuleiro mantem-se inalterado.

- tabuleiro-remove-moeda-posicao!: tabuleiro x posição → {}

  Este modificador recebe um tabuleiro e uma posição, e remove a moeda que se encontra no tabuleiro na posição recebida. Esta função não retorna nada, mas **altera** o tabuleiro recebido removendo a moeda em questão. Caso não exista nenhuma moeda na posição recebida, esta função não faz nada.

2.1.4 Tipo Jogo

Este tipo representa o estado de uma instância do jogo Moedas e Fios. Para além do estado do tabuleiro, este tipo guarda informação acerca do número de pontos obtidos para cada jogador, do próximo jogador a jogar, e do histórico de todas as acções (i.e. fios removidos) feitas desde o início do jogo. É também responsável por efectuar as jogadas e actualizar o estado de acordo com a jogada escolhida, garantindo um estado consistente de jogada para jogada. As operações básicas associadas a este tipo são:

- cria-jogo: tabuleiro → jogo

  Este construtor recebe um tabuleiro e retorna o jogo que corresponde ao estado inicial do jogo para o tabuleiro recebido. Ou seja, ambos os jogadores começam com 0 pontos, o próximo jogador a jogar é o jogador 1, e ainda nenhuma jogada foi efectuada.

- copia-jogo: jogo → jogo

  Este construtor recebe um jogo, e devolve um **novo** jogo cujo conteúdo é o mesmo do jogo recebido. O jogo retornado tem que ser uma nova cópia, i.e. se o jogo original for alterado, o jogo retornado não pode mudar.

- jogo-tabuleiro: jogo → tabuleiro

  Este selector recebe um jogo e retorna o tabuleiro com o estado do jogo.

- jogo-jogador: jogo → inteiro

  Este selector recebe um jogo e retorna um inteiro (1 ou 2) que indica qual o jogador a jogar.

- jogo-pontos-jogador1: jogo → inteiro

  Este selector recebe um jogo e retorna um inteiro que indica o número de pontos acumulados pelo jogador1 ao longo do jogo.

- jogo-pontos-jogador2: jogo → inteiro

  Este selector recebe um jogo e retorna um inteiro que indica o número de pontos acumulados pelo jogador2 ao longo do jogo.

- jogo-historico-jogadas: jogo → lista de inteiros

  Este selector recebe um jogo e retorna uma lista com todas as jogadas efectuadas desde o estado inicial do jogo. Como as jogadas correspondem a remover fios, este procedimento retorna uma lista com os identificadores (inteiros) dos fios removidos. Os elementos da lista devem estar ordenados pela ordem da jogada, ou seja, o 1.º elemento da lista deve corresponder ao primeiro fio removido e assim sucessivamente.

- jogo-aplica-jogada!: jogo x inteiro→ {}

  Este modificador é responsável por efectuar uma jogada do jogo Moeda e Fios. Como as jogadas correspondem a remover fios, este procedimento recebe um jogo e o identificador do fio a remover do tabuleiro. Este procedimento não retorna nada, mas **altera** o jogo recebido de modo a aplicar a jogada feita. Primeiro remove-se o fio. Se o fio removido corresponder ao último fio de uma ou mais moedas, essas moedas deverão ser também removidas do tabuleiro, e os seus pontos atribuídos ao jogador que fez a jogada. Nesta situação, o jogador joga novamente. Caso não seja o ultimo fio de nenhuma moeda, apenas se remove o fio, e passa-se o turno ao próximo jogador.

- jogo-terminado-p: jogo → lógico

  Este reconhecedor recebe um jogo e verifica se o jogo recebido corresponde a um jogo que terminou. Um jogo termina quando não existem mais fios para serem removidos.

2.1.5 Tipo Problema

Para além dos tipos identificados acima é necessário também definir o tipo Problema, que representa um problema de procura adversária. No entanto, ao contrário dos outros tipos onde é deixado ao critério do

aluno qual a implementação a usar, aqui o tipo problema deverá ser implementado como uma estrutura[2], com os seguintes campos:

- estado-inicial – corresponde ao estado inicial do problema a resolver
- jogador – função que recebe um estado e retorna o id do jogador que vai jogar no estado actual
- accoes – função que recebe um estado e devolve uma lista de acções válidas para esse estado
- `resultado` – função que recebe um estado e uma acção e retorna o estado que resulta de aplicar a acção no estado.
- teste-corte-p – predicado que recebe um estado, a profundidade do estado na árvore. Este predicado testa se o estado recebido corresponde a um estado terminal (i.e. o jogo terminou), ou se a profundidade máxima permitida da árvore foi atingida.
- funcao-avaliacao – função que recebe um estado e o id de um jogador, e retorna uma estimativa da utilidade esperada do estado para o jogador. Se o estado recebido for um estado terminal, esta função deve retornar a utilidade exacta do estado para o jogador.
- `historico-accoes` – função que recebe um estado do jogo e que retorna uma lista com todas as acções executadas desde o estado inicial até o estado recebido. As acções estão ordenadas pela ordem de execução.
- `chave-equivalencia` – função que recebe um estado e retorna uma chave do estado. A chave retornada pode ser um objecto qualquer desde que satisfaça as seguintes restrições: dois estados equivalentes devem gerar a mesma chave; dois estados não equivalentes devem gerar chaves distintas.

Esta estrutura irá ser usada como argumento dos algoritmos minimax, como será descrito mais à frente. Note que as funções historico-accoes e chave-equivalencia só serão necessárias para implementação de funcionalidades mais avançadas do algoritmo minimax.

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

Estas funções estão relacionadas com a implementação dos métodos genéricos de um problema, aplicados ao jogo Moedas e Fios:

- accoes: jogo → lista de inteiros

  Esta função recebe um estado, que no caso deste problema é um jogo, e devolve uma lista com todas as acções válidas para esse estado. No problema do Moedas e Fios, uma acção corresponde a remover um fio, por isso a acção é representada apenas pelo identificador do fio a remover. Ou seja, esta função retorna uma lista com os id's dos fios que são possíveis remover no jogo recebido.

[2] Usando a primitiva defstruct.

- resultado: jogo x inteiro → jogo

  Esta função recebe um jogo, e um inteiro que corresponde ao identificador do fio a remover, e retorna um novo jogo que resulta de fazer a acção (i.e. remover o fio). **Muito importante:** esta função **não pode alterar** o jogo recebido.

- teste-terminal-p: jogo x inteiro x inteiro → lógico

  Esta função recebe um jogo, e um inteiro correspondente à profundidade do estado. No entanto ignora o 2.º argumento completamente[3], e retorna T se e só se o jogo recebido for um jogo terminado. Retorna NIL caso contrário.

- utilidade: jogo x inteiro → inteiro

  Esta função recebe um jogo correspondente a um estado terminal, e um identificador de um jogador (1, ou 2) e retorna a utilidade exacta do jogo para o jogador em questão.

2.2.2 Minimax e Jogador simples

As funções descritas nesta subsecção correspondem às versões mais simples a implementar para o algoritmo minimax, e para o jogador automático. Estas funções deverão ser implementadas na 1.ª entrega, e uma vez implementadas, podem experimentar jogar contra o jogador minimax-simples. No entanto, apenas conseguirão jogar os primeiros tabuleiros mais simples, pois nos tabuleiros maiores facilmente ficarão sem paciência para esperar algumas horas/dias para ser escolhida uma jogada.

- minimax: problema x inteiro → accao x inteiro x inteiro

  Esta função recebe um problema (estrutura definida acima), e um inteiro (1 ou 2) que indica qual dos jogadores é o jogador max. Corresponde a executar o algoritmo minimax **sem cortes** alfa-beta. Retorna três coisas[4]: a acção escolhida como a melhor acção a ser executada pelo jogador max, o valor minimax dessa mesma acção, e o número de nós folha visitados na árvore. **Muito importante**: o algoritmo minimax não pode ser implementado tal como está definido no Livro da cadeira. A versão descrita no Livro assume que cada jogador joga alternadamente, o que não acontece em todos os problemas como é o caso do jogo Moedas e Fios (um jogador pode jogar várias vezes seguidas). É necessário adaptar o algoritmo de modo a que permita lidar com esta situação. Para além disto devem também ter o cuidado do algoritmo minimax ser independente do problema, ou seja deverá funcionar para este problema do jogo Moedas e Fios, mas deverá funcionar também para qualquer outro problema[5].

- jogador-minimax-simples: jogo x inteiro x inteiro → accao

  Esta função recebe um jogo, um inteiro (1 ou 2) que indica qual o jogador a jogar, e um inteiro que indica o tempo limite (em segundos) para o jogador tomar uma decisão. Esta função ignora o tempo limite recebido e implementa um jogador automático que decide qual a melhor acção

[3] Para não obterem um warning na compilação devem fazer "(declare (ignore … ….))" dos argumentos recebidos que querem ignorar. Isto deve ser feito na 1.ª linha do corpo da função.

[4] Para ver como uma função pode devolver mais do que um valor, consultem a primitiva `values` do Lisp.

[5] Portanto, é desaconselhado o uso de variáveis globais, e usar/aceder directamente a funções específicas deste jogo. Tudo o que precisarem vai estar dentro da estrutura problema recebida.

para o jogador usando o algoritmo minimax. Este jogador deve usar o algoritmo minimax **sem cortes** alfa-beta, e **sem** qualquer **limite** de profundidade ou tempo, percorrendo todo o espaço de estados. A função de avaliação corresponde simplesmente a calcular a utilidade de um estado.

2.2.3 Minimax e Jogadores mais avançados

Nesta subsecção serão descritas algumas das funções a implementar para a 2.ª entrega do projecto. Correspondem a versão mais elaboradas do algoritmo minimax e do jogador automático.

- minimax-alfa-beta: problema x inteiro → accao x inteiro x inteiro

  Esta função recebe um problema, e um inteiro (1 ou 2) que indica qual o jogador max. Corresponde a executar o algoritmo minimax **com cortes** alfa-beta. Retorna três coisas: a acção escolhida como melhor acção a ser executada pelo jogador max, o valor minimax dessa mesma acção, e o número de nós folha visitados na árvore. Deverão ter os mesmos cuidados que na implementação da função minimax descrita na subsecção anterior.

- jogador-minimax-vbest: problema x inteiro x inteiro → accao, inteiro

  Esta função recebe um problema, um inteiro (1 ou 2) que indica qual o jogador max, e um inteiro que indica o número de segundos que o jogador tem para escolhar a jogada. Esta função tem de garantir que é devolvida uma jogada **dentro do tempo limite** especificado. No entanto, se não for necessário gastar o tempo total para tomar uma decisão (por exemplo o problema é muito pequeno), devem retornar a jogada o mais rápido possível. Deverá utilizar a vossa **melhor versão** do algoritmo minimax, e a **melhor função de avaliação** desenvolvida. Esta será a função utilizada para testar a qualidade final do vosso jogador automático, e será também a função utilizada no Torneio do Projecto.

Para além destas duas funções explicitamente definidas (que serão testadas automaticamente), na 2.ª fase do projecto terão que implementar mais funções. Provavelmente será necessária a implementação de outras variantes do algoritmo minimax, com funcionalidades adicionais. Deverão também implementar outras versões do jogador (e.g. jogador-minimax-v1[6], jogador-minimax-v2, etc) que vos permita comparar a *performance* de um jogador com ou sem cortes alfa-beta (em termos de qualidade das escolhas, profundidade máxima atingida, nós folha visitados, etc), e comparar também a utilização de diferentes funções de avaliação.

Como irão perceber ao longo do projecto, a função de avaliação tem um impacto enorme na qualidade das escolhas feitas pelo jogador automático. Deverão implementar 2 a 3 funções de avaliação distintas (ou mais se assim o entenderem). A função **random não é** considerada como uma função de avaliação **válida**.

2.3 Estudo e análise dos algoritmos minimax e funções de avaliação

Na parte final do projecto, é suposto os alunos compararem as diferentes variantes dos algoritmos minimax implementados (ex: com e sem cortes), 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.

[6] Estas versões deverão também ter limite de tempo.

É também importante comparar as diferentes funções de avaliação implementadas, e perceber o impacto delas na qualidade do jogador automático obtido. Deverão comparar as funções num conjunto significativo de exemplos (e não necessariamente apenas nos exemplos fornecidos pelo corpo docente).

Os resultados dos testes efectuados deverão ser usados para escolher a melhor variante do algoritmo minimax e a melhor função de avaliação a ser usada no jogador-minimax-vbest. 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); como é que implementaram o minimax de modo a permitir um jogador jogar múltiplas vezes; que outro tipo de variantes do algoritmo minimax foram implementadas e como foram implementadas; como foi implementado o mecanismo de limitação de tempo no jogador minimax; quais as funções de avaliação implementadas e a ideia por detrás delas, 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 Interface jogo Moedas e Fios e ficheiros fornecidos

Foram fornecidos dois ficheiros juntamente com o enunciado. O 1.º ficheiro, interface-moedas.lisp corresponde à implementação dos métodos que desenham um jogo/tabuleiro no ecrã e de métodos usados para jogar um jogo Moedas e Fios. O 2.º ficheiro contem 10 exemplos de tabuleiros (t1,t2,t3,t4,t5,t6,t7,t8,t9,t10) que poderão ser usados para testar o vosso código. No entanto, o código fornecido nestes ficheiros só funcionará correctamente depois de implementados os tipos de informação especificados. Para terem acesso às funções e aos tabuleiros de exemplo, deverão copiar os ficheiros fornecidos para a mesma directoria onde se encontra o vosso projecto, e deverão colocar as seguintes instruções a seguir às vossas definições dos Tipos de Informação:

```
(load (compile-file "interface-moedas.lisp"))

(load (compile-file "exemplos.lisp"))
```

Iremos descrever brevemente as funções mais relevantes fornecidas no ficheiro de interface. A função desenha-tabuleiro é usada para desenhar um tabuleiro no ecrã. Isto é feito usando caracteres ASCII. Por exemplo se desenharem o tabuleiro 1 vão obter o seguinte *output*:

```
> (desenha-tabuleiro t1)

================================
||   (5)---01--(3)             ||
||    | \     / | \            ||
||    |  03     |  07          ||
||    02   x    05   \         ||
||    |  04     |              ||
||    | /     \ |       \      ||
||   (1)---06--(1)---08--(6)    ||
================================
NIL
```

As moedas são representadas dentro de parênteses, e cada fio tem um id (numero entre 01 e 99).

A função desenha-jogo é semelhante à função desenha-tabuleiro, no entanto imprime mais informação, em particular a pontuação actual para cada jogador.

A função joga executa um jogo completo a partir de um tabuleiro recebido. Recebe duas funções que correspondem aos jogadores que vão jogar o jogo. Podemos passar um quarto argumento opcional que indica o tempo limite (em segundos) que deve ser gasto por cada jogador para tomar a decisão. Se não for especificado nenhum tempo limite, o valor por omissão são de 30 segundos. Se quisermos ter um jogador humano, podemos passar como argumento o procedimento *jogador-humano*, que está incluído no ficheiro de interface fornecido. Por exemplo, a seguinte instrução irá correr um jogo entre dois jogadores humanos.

```
> (joga t2 #'jogador-humano #'jogador-humano)
==============================
||  (5)---01--(3)---05--(8)  ||
||   |         |         |   ||
||   |         |         |   ||
||   02        04        07  ||
||   |         |         |   ||
||   |         |         |   ||
||  (1)---03--(1)---06--(6)  ||
||   |         |         |   ||
||   |         |         |   ||
||   08        09        10  ||
||   |         |         |   ||
||   |         |         |   ||
||  (2)---11--(4)---12--(7)  ||
==============================
Jogador 1:   0 moedas     Jogador 2:   0 moedas

Jogador 1, escolha um fio a remover: a
O valor introduzido nao e um inteiro valido.
Jogador 1, escolha um fio a remover: 34
Jogada invalida. O fio inserido nao existe.
Jogador 1, escolha um fio a remover: 5

==============================
||  (5)---01--(3)        (8)  ||
||   |         |         |   ||
||   |         |         |   ||
||   02        04        07  ||
||   |         |         |   ||
||   |         |         |   ||
||  (1)---03--(1)---06--(6)  ||
||   |         |         |   ||
||   |         |         |   ||
||   08        09        10  ||
||   |         |         |   ||
||   |         |         |   ||
||  (2)---11--(4)---12--(7)  ||
==============================
Jogador 1:   0 moedas     Jogador 2:   0 moedas
```

```
Jogador 2, escolha um fio a remover: 7

================================
||   (5)---01--(3)             ||
||    |          |              ||
||    |          |              ||
||    02         04             ||
||    |          |              ||
||    |          |              ||
||   (1)---03--(1)---06--(6)    ||
||    |          |          |    ||
||    |          |          |    ||
||    08         09         10   ||
||    |          |          |    ||
||    |          |          |    ||
||   (2)---11--(4)---12--(7)    ||
================================
Jogador 1:   0 moedas     Jogador 2:   8 moedas

Jogador 2, escolha um fio a remover:
```

Neste excerto de um jogo entre dois jogadores humanos, podemos ver que se introduzirmos algo que não seja um número ou um número que não corresponda a um fio do tabuleiro, o sistema irá pedir uma nova jogada. Podemos também ver a situação em que o Jogador 2 joga de novo por ter ficado com uma moeda.

Para colocarmos um jogador humano a jogar contra um jogador automático, basta usar a função desejada do jogador automático. Por exemplo, assumindo que a função jogador-minimax-vbest está definida, podemos fazer:

```
(joga t2 #'jogador-minimax-vbest #'jogador-humano)
```

Ou até mesmo, colocar um jogador automático contra outro automático, com o máximo de 15 segundos para cada um:

```
(joga t2 #'jogador-minimax-vbest #'jogador-minimax-v1 15)
```

Em relação aos tabuleiros de teste, vamos permitir que os grupos partilhem outros tabuleiros adicionais de teste, se assim o entenderem. Aliás, se criarem algum tabuleiro de teste que achem interessante, enviem-no por email ao corpo docente. Iremos analisar as submissões de tabuleiros recebidas, e se forem de facto interessantes poderão ser usados no torneio ou até mesmo disponibilizados publicamente para todos os alunos.

4 Game On: Torneio do Projecto

Vai ser realizado um torneio do Jogo Moedas e Fios para determinar quais os melhores jogadores automáticos criados pelos alunos da cadeira de IA. Para poderem participar no torneio, a vossa implementação tem que passar todos os testes de execução referentes às funções e tipos de dados definidos neste enunciado.

O torneio vai ser realizado em duas fases: a fase de qualificação e a fase final. Na fase de qualificação, serão escolhidos os 4 melhores projectos da Alameda e os 4 melhores projectos do Tagus para passarem à fase final. A organização exacta da fase de qualificação vai depender do número de projectos submetidos, mas a ideia é os projectos serem inicialmente organizados por grupos em cada campus, onde um jogador vai jogar contra todos os jogadores do mesmo grupo. No fim, por cada grupo, os dois melhores projectos (i.e. com melhor pontuação total dos confrontos efectuados) passam para uma 2.ª ronda eliminatória para decidir os 4 melhores projectos do campus.

Na fase final, os projectos qualificados irão ser inicialmente distribuídos da seguinte maneira: o 1.º classificado do Tagus irá defrontar o 4.º da Alameda; o 2.º classificado do Tagus irá defrontar o 3.º da Alameda; o 1.º classificado da Alameda irá defrontar 4.º do Tagus; o 2.º classificado da Alameda irá defrontar o 3.º do Tagus. Nas meias finais os confrontos serão sorteados. O terceiro lugar é determinado pelo confronto entre os dois grupos que perderem nas meias finais.

Os 3 primeiros classificados no torneio 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

A função que vai ser usada como o vosso jogador automático no torneio é a vossa função jogador-minimax-vbest. Na fase de qualificação o vosso jogador irá ter **15 segundos** para tomar uma decisão. Se levar mais do que o tempo permitido para decidir a jogada, perde automaticamente o jogo, ficando o adversário com o máximo de pontos possíveis no tabuleiro. Na fase final do torneio o vosso jogador irá ter **30 segundos** para tomar uma decisão.

No confronto directo entre um jogador j_1 contra outro jogador j_2 irão ser usados um conjunto de tabuleiros heterogéneos, que podem ir desde alguns dos exemplos fornecidos pelo corpo docente, a outros tabuleiros usados nos testes de avaliação, ou até mesmo tabuleiros interessantes fornecidos pelos alunos. Para cada tabuleiro serão feitos dois jogos, pois a ordem pela qual os jogadores jogam tem muita influência no resultado final. No 1.º jogo, o jogador j_1 joga como primeiro jogador e o jogador j_2 como segundo jogador. No 2.º jogo, inverte-se a ordem dos jogadores. A pontuação obtida num tabuleiro para um jogador corresponde à soma da pontuação dos dois jogos. A pontuação obtida num jogo corresponde à pontuação do jogador subtraída da pontuação do jogador adversário. A pontuação obtida num confronto corresponde à soma das pontuações obtidas para cada tabuleiro jogado no confronto. O jogador com maior pontuação num confronto directo ganha esse confronto.

É importante frisar que a implementação dos tipos (fio,tabuleiro,jogo,etc) usada no torneio vai ser fornecida pelo corpo docente[7]. Assim sendo, devem ter o cuidado para não quebrarem as barreiras de abstração, e não se devem focar demasiado na vossa implementação dos tipos para criarem um jogador mais eficiente/inteligente. Como sugestão, deixamos aqui a ideia de que apostar em boas variantes do algoritmo minimax e em boas funções de avaliação/heurísticas poderá valer a pena. No entanto, se quiserem ir por esse caminho, podem sempre pegar no jogo recebido como argumento da função jogador-

[7] Não seria fácil usar as duas implementações distintas dos tipos dos dois projectos ao mesmo tempo, e pensamos que esta é a melhor opção.

minimax-vbest e criar um estado mais elaborado (i.e. com mais informação) ou mais eficiente a partir dele, e usar esse estado para correr com o vosso algoritmo minimax.

5 Entregas, Prazos, Avaliação e Condições de Realização

5.1 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.

5.1.1 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 às **secções 2.2.1** e **2.2.2**. A entrega desta primeira parte será feita por via electrónica através do sistema ***Mooshak***, até às **23:59** do dia **11/11/2013**. Depois desta hora, não serão aceites projectos sob pretexto algum[8].

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.

5.1.2 2.ª Entrega

Na segunda entrega do projecto os alunos devem implementar o resto das funcionalidades descritas no enunciado (ver **secção 2.2.3**), incluindo as várias variantes do algoritmo minimax, e as funções de avaliação pedidas. Deverão também já nesta fase fazer os testes que permitirão escolher qual melhor o algoritmo/função de avaliação (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 **02/12/2013**. 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.

5.1.3 3.º 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** do dia **9/12/2013**.

5.2 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).

[8] 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.

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 automático criado (quão bom ou mau é o jogador a jogar o jogo Moedas e Fios). A avaliação da qualidade do jogador é feita confrontando o jogador implementado pela função jogador-minimax-vbest com vários jogadores (fáceis, médios e difíceis) criados pelo corpo docente. O confronto é feito nos mesmos moldes do mecanismo usado no torneio, com a diferença que a implementação dos tipos usada será a dos alunos.

A 2.ª componente vale 1 valor e 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.

5.3 Condições de realização

O código desenvolvido deve compilar em CLISP 2.49 sem qualquer "warning" ou erro. Todos os testes efectuados automaticamente, e mesmo o torneio, 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 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 minimax e as funções de avaliação. 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 automáticamente 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, 28 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.[9]

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

Projectos muito semelhantes serão considerados cópia e rejeitados. A detecção de semelhanças entre projectos será realizada utilizando *software* especializado[10], 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.

[9] 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.
[10] Ver http://theory.stanford.edu/~aiken/moss/