Grupo Nº 003

Inteligência Artificial

1.º Semestre 2013/2014

Moedas e Fios

Relatório de Projecto

Índice

1 Implementação Tipos e Problema

1.1 Tipos Abstractos de Informação

Tipo Posição:

O tipo posição é utilizado para representar uma posição do tabuleiro (através da sua linha e coluna). Optou-se por simplicidade e flexibilidade de criar apenas uma estrutura com 2 campos:

- `linha` - Inteiro que representa a linha do tabuleiro;
- `coluna` - Inteiro que representa a coluna do tabuleiro.

Em relação às funções implementadas:

- `cria-posicao(x, y)` - Recebe 2 inteiros `x` e `y` e retorna uma estrutura `posicao`;
- `posicoes-iguais-p(p1 p2)` - Duas posições `p1` e `p2` são idênticas se e só se as suas coordenadas forem idênticas. Como tal, são feitas comparações campo a campo. Uma possível alternativa seria usar directamente a função de teste `equalp`.

As restantes funções abaixo indicadas foram automaticamente implementadas pelo *CLISP* quando definimos a estrutura `posicao`:

- `posicao-linha(posicao)` – retorna a linha da `posicao`;
- `posicao-coluna(posicao)` – retorna a coluna da `posicao`;
- `posicao-p(posicao)` – função que verifica se o argumento é uma estrutra `posicao`.

Tipo Fio:

O tipo fio é utilizado para representar um fio do tabuleiro que liga um par de moedas. Cada fio é identificado unicamente por um inteiro superior a 0 e cada fio, contém em cada uma das pontas uma posição. Como tal, optámos por implementar o tipo fio contendo:

- `id` - Identificador único do fio superior a 0;
- `origem` - Posição de origem do fio;
- `destino` - Posição de destino do fio.

A única função implementada explicitamente foi o `cria-fio(i pos-orig pos-dest)` que recebe o id do fio e duas posições e cria a estrutura do tipo fio. As restantes funções abaixo indicadas foram automaticamente implementadas pelo *CLISP* quando definimos a estrutura fio:

- `fio-id(fio)` – retorna o id do `fio;`
- `fio-origem(fio)` – retorna a posição de origem do `fio;`
- `fio-destino(fio)` – retorna a posição de destino do `fi;`.
- `fio-p(obj)` – função de teste que verifica se variável `obj` é um fio.

Tipo Tabuleiro:

O tipo de dados `tabuleiro` é usado para representar o estado de um tabuleiro do jogo Moedas e Fios, contendo:

- ➔ `matriz` - Uma matriz de inteiros com x número de linhas e y número de colunas. Inicializado com o valor `NIL`;
- ➔ `fios` - Lista de fios do tabuleiro;
- ➔ `quant-moedas` - Inteiro representando o total de moedas disponíveis no tabuleiro;
- ➔ `total-fios` - Inteiro representando o número total de fios no tabuleiro desde o início do jogo.

Optou-se por esta implementação específica de um tabuleiro com as estruturas de dados acima mencionadas de forma a reduzir a complexidade durante a leitura e escrita de dados. Mais especificamente, optou-se por um vector bidimensional cujos acessos de leitura e escrita são feitos com complexidade O(1).

Optou-se por uma lista de fios para poder armazenar todos os fios do tabuleiro, no entanto, a complexidade de leitura e de escrita são nos piores casos O(N). Em alternativa, poder-se-ia ter implementado hashmaps cuja chave seria o id do fio e o valor seria a estrutura fio reduzindo a complexidade para O(1), o que poderia fazer diferença dado que são leituras e escritas que ocorrem muito frequentemente.

A variável `quant-moedas` tem como propósito evitar que a cada chamada da função `tabuleiro-total-moedas` ter que percorrer a matriz linha a linha, coluna a coluna (com complexidade $O(N^2)$) e somar os pontos onde existem moedas. Como tal, basta actualizar o valor e aceder directamente à informação contida na estrutura.

À semelhança da da variável `quant-moedas`, é guardado o número de fios do tabuleiro para para evitar percorrer a lista de fios e contar o número de fios de modo a gerar um novo id durante a inserção de um novo fio no tabuleiro.

Em relação às funções implementadas:

- ➔ `cria-tabuleiro(x y)`**:** É criado um tabuleiro com uma matriz de `x` linhas, `y` colunas, uma lista vazia de fios, e são inicializadas as variáveis `quant-moedas` e o `total-fios` a 0.
- ➔ `copia-tabuleiro(tab)`**:** É criada uma cópia do tabuleiro onde são feitas cópias explícitas (i.e., com alocação de memória) da matriz e da lista de fios.
- ➔ `tabuleiro-fio-com-id(tab id)`**:** Dado que um fio tem como identificador único o id, é iterada a lista de fios até encontrar um fio com o `id` dado como argumento retornando o mesmo. Tal como referido acima, caso optássemos tabelas de dispersão verificaríamos que os acessos seriam mais rápidos e o código seria mais simples utilizando apenas funções *built-in* no *Common Lisp*.
- ➔ `tabuleiro-fios-posicao(tab pos)`**:** Um fio só está ligado a uma posição `pos` se a posição de origem ou de destino for idêntica à posição dada como argumento. Como tal, é percorrida a lista de fios do tabuleiro e verifica-se se a posição de origem do fio ou a posição de destino do fio em análise satisfaz o predicado `posições-iguais-p` com a variável `pos` dada

como argumento. Caso satisfaça o predicado, adiciona-se à lista auxiliar que é retornada no final da função.

- ➔ `tabuleiro-moeda-posicao(tab pos)`**:** É retornado directamente o valor indicado na matriz na posição `pos` dada como argumento. Caso não haja moeda, o valor retornado será `NIL`, pois a matriz é inicializada a `NIL`.
- ➔ `tabuleiro-total-moedas(tab)`**:** Tal como explicado anteriormente, utilizamos uma variável contida no tabuleiro que indica o número total de moedas disponíveis no tabuleiro. Como tal, esta função apenas retorna o valor armazenado no tabuleiro `tab` dado como argumento.
- ➔ `tabuleiro-adiciona-fio!(tab pos1 pos2)`**:** Quando se adiciona um fio ao tabuleiro `tab` dado como argumento é necessário gerar um novo id único e, como é alocado no tabuleiro a contagem do numero total de fios, o novo id será o número total de fios existentes até ao momento acrescentado de um valor. Por fim, basta juntar o novo fio no fim da lista de fios já existente no tabuleiro.
- ➔ `tabuleiro-adiciona-moeda-posicao!(tab pos val)`**:** É alterado directamente na matriz o valor associado à posição `pos` dada como argumento. Como usamos uma variável que contabiliza o número de total de moedas disponíveis num tabuleiro actualizamos esse mesmo valor.
- ➔ `tabuleiro-remove-fio-com-id!(tab id)`**:** Enquanto é percorrida lista de fios associados ao tabuleiro `tab` dado como argumento, é construída uma lista com os fios até ao momento analisada. Só quando é encontrado o fio com o `id` dado como argumento é que devolvemos o resto da lista.
- ➔ `tabuleiro-remove-moeda-posicao!(tab pos)`**:** Caso houver uma moeda associada à posição `pos` no tabuleiro `tab`, é decrementado da variável que controla o numero total de moedas disponíveis no tabuleiro. No fim, é actualizada a matriz com `NIL` na posição `pos` indicando deixou de estar disponivel uma moeda.

As seguintes funções são automaticamente implementadas quando se gera a estrutura `tabuleiro` com os campos indicados no início.

- ➔ `tabuleiro-matriz(tab)` - Devolve a matriz de moedas;
- ➔ `tabuleiro-fios(tab)` - Devolve a lista de fios;
- ➔ `tabuleiro-quant-moedas(tab)` - Devolve o número de total de moedas no tabuleiro;
- ➔ `tabuleiro-total-fios(tab)` - Devolve o numero de fios gerados desde a criação do tabuleiro.

Jogo:

O tipo Jogo representa o estado de uma instância do jogo Moedas e Fios. É guardado o estado do tabuleiro e a informação acerca do número de pontos obtidos para cada jogador, do próximo jogador a jogar, e do histórico de todas as acções realizadas desde o início do jogo. Não foram adicionados campos à estrutura de dados para além das mencionadas no enunciado.

Em relação às funções implementadas:

`cria-jogo(tab)`: É criado um novo jogo no tabuleiro dado como argumento com o jogador 1 a ser o primeiro a jogar e a começar com pontuação 0 (tal como o jogador 2). O histórico encontra-se vazio.

`copia-jogo(jogo)`: É criado um jogo cujos elementos são cópias de cada elemento da estrutura do `jogo` dado como argumento.

`alterna-jogador(jogo)`: Função auxiliar utilizada para alternar entre o jogador 1 e o jogador 2 da variável `jogo`.

`adiciona-pontos-jogador(jogo pontos)`: Função auxiliar utilizada para adicionar pontos ao jogador actual do `jogo` dado como argumento.

`jogo-aplica-jogada!(jogo id)`: Dado um `id` de um fio existente no tabuleiro do `jogo`, guardamos as informações acerca do fio, eliminamos-lo da lista de fios do tabuleiro para de seguida analisar quantos pontos serão ganhos. Esta análise é feita através da utilização de uma função auxiliar chamada `actualiza-posicao(tab pos)` que verifica se depois de retirado o fio ainda existe algum associado à moeda, caso acontecer devolve o valor da moeda, caso contrário devolve 0. Deste modo, os pontos ganhos será a soma das chamadas à função `actualiza-posicao` com a posição de origem do fio e a chamada com o destino do fio. Por fim é actualizado os pontos do jogador actual, é actualizado o histórico e alternamos os jogadores caso foram ganhos pontos.

`jogo-terminado-p (jogo)`: Um jogo só está terminado se não houver mais fios para retirar do tabuleiro. Em alternativa podia-se verificar se o total-moedas do tabuleiro era 0.

As seguintes funções são automaticamente implementadas quando se gera a estrutura `jogo` com os campos indicados no início.

- ➔ `jogo-tabuleiro(jogo)`: Devolve o tabuleiro do jogo;
- ➔ `jogo-historico-jogadas(jogo)`: Devolve o historico de jogadas;
- ➔ `jogo-jogador(jogo)`: Devolve o jogador actual;
- ➔ `jogo-pontos-jogador2(jogo)`: Devolve a pontuação do jogador 2;
- ➔ `jogo-pontos-jogador2(jogo)`: Devolve a pontuação do jogador 1.

Problema:

O tipo Problema, representa um problema de procura adversária. Esta estrutura é idêntica à mencionada no enunciado do presente projecto cujas funções por nós implementadas são descritas na secção 1.2. É de salientar que não foi utilizado o campo chave-equivalência nem o histórico de acções.

Jogo-equivalente:

A estrutura `jogo-equivalente` é utilizado para representar estados equivalentes quando usamos tabelas de transposição. Dois jogos são equivalentes se e só se forem idênticos: o jogador actual, as pontuações de cada jogador e a lista de fios. Como tal, a tabela de transposição é representado através de uma tabela de dispersão cuja chave é a estrutura `jogo-equivalente`

como função de teste o `equalp` e o valor obtido será o minimax calculado noutros nós da mesma árvore em análise.

As funções implementadas para esta estrutura de dados foram as seguintes:

- `cria-jogo-equivalente(jogo)`: Cria uma versão simplificada do jogo utilizada.
- `cria-tabela-transposicao()`: Retorna uma hashmap cuja função de teste é o `equalp` que permite verificar se duas estruturas são idênticas.
- `procura-estado-minimax(jogo hash)`: Procura na tabela de transposição hash pelo valor do minimax retornado para o jogo dado como argumento. Devolve NIL caso não tenha encontrado.
- `adiciona-estado-minimax!(jogo minimax hash)`: Dado um jogo, cria um jogo equivalente e adiciona uma entrada na hash com o valor minimax dado como argumento.

A utilização das tabelas de transposição é explicada com mais detalhe no ponto 2.5.

1.2 Funções específicas do problema

As funções específicas do tipo problema implementadas foram:

- `accoes(jogo)`: Devolve uma lista de fios, iterando a lista todas de fios retirando apenas o id. É feito iterativamente de modo a devolver uma lista decrescente de ids.
- `resultado(jogo id)`: É feita uma cópia do jogo e é aplicado sobre a cópia a função jogo-aplica-jogada! com o id do fio.
- `teste-terminal-p(jogo profundidade)`: É feita uma chamada directa ao `jogo-terminado-p`.
- `utilidade(jogo jogador)`: Devolve a diferença de pontos entre o jogador 1 e o jogador 2. A variável `jogador` irá apenas indicar quem será o jogador que estará a tentar maximizar a diferença de pontos.

Todos os acessos aos campos `estado-inicial`, `jogador`, `accoes`, `resultado`, `teste-corte-p`, `funcao-avaliacao`, `historico-accoes` e `chave-equivalencia` são feitos utilizando as funções implementadas automaticamente pelo *Common Lisp* assim que definida a estrutura Problema.

2 Algoritmos Minimax, Jogador e Variantes

2.1 Minimax com múltiplas jogadas por jogador

De forma a permitir que um jogador possa jogar múltiplas vezes seguidas, foi colocado dentro do minimax, uma verificação de qual o jogador a realizar a jogada em cada estado (jogo).

Esta verificação assenta na chamada à função `jogo-jogador` para o estado actual, que devolve tal como o nome indica o jogador para esse estado. Como o jogador-max é passado como argumento da função é apenas necessário comparar o jogador do estado actual com o jogador-max passado como argumento, de forma a fazermos o máximo ou o minimo dos valores minimax obtidos para cada sucessor de um estado.

2.2 Mecanismo de limitação de tempo no jogador automático

De modo a ser possível retornar uma jogada dentro do tempo limite, recorremos a uma técnica de comparação de intervalos de tempo, baseada na obtenção do tempo actual com a função `get-internal-real-time` do *Common Lisp*.

No entanto, é dada uma margem de tempo de modo a que a função `minimax-vbest` retorne a melhor jogada em tempo útil. Para tal, houve necessidade de estimar uma constante empírica que iria representar qual seria seria a percentagem do tempo necessária retornar a melhor jogada calculada após se detectar que o tempo estava a chegar ao fim. Verificou-se, utilizando diferentes computadores, submissões ao mooshak e diferentes jogos com diferentes tempos limite, que seria necessário dedicar 5% do tempo disponível para a função retornar uma jogada válida com segurança. Dessa forma, decidimos que a função minimax iria usar 95% do tempo disponível para realizar o algoritmo minimax.

É tomado o cuidado comparando em cada estado se a diferença entre o tempo actual e o tempo obtido no inicio da execução (instante inicial inicializado no início do `minimax-vbest`) é inferior ao tempo limite permitido de execução, caso seja, continuamos para o estado seguinte, caso isso não se verifique, é retornada a melhor jogada guardada correspondente á jogada selecionada da iteração anterior.

Optámos por realizar esta comparação em cada estado, e não em cada iteração, pois não teríamos forma de garantir, por muito boa que fosse a nossa estimativa, se o tempo restante seria suficiente para o algoritmo correr mais um nível da arvore de procura, e cairíamos na situação em que o tempo se acabaria e não retornaríamos uma jogada. Por outro lado, a verificação em cada estado, embora menos eficiente garante sempre o retorno, dentro do tempo, da melhor jogada.

2.3 Variante 1 do algoritmo minimax

A primeira variação do algoritmo minimax com cortes alfa beta, foi a preparação do algoritmo para utilização em profundidade iterativa. Tal como descrito no ponto anterior, sabendo que haveria possibilidade de não percorrer todos os nós terminais da árvore de procura dado que teríamos um tempo limitado de execução, onde antes apenas se verificava se estávamos num estado terminal agora essa verificação encontra-se dividida em duas condições: uma caso atinjamos a profundidade máxima dessa iteração, caso em que retornaríamos o valor da heurística para esse estado, e outra caso atinjamos um nó terminal, caso em que retornaríamos a utilidade para esse estado.

Como no `minimax-alfa-beta` original, são calculados os sucessores para todos os estados desse nível e antes de iterar a lista resultante.

2.4 Variante 2 do algoritmo minimax

A segunda versão do minimax, conta apenas com uma alteração da forma de aplicação da jogada ao estado que permitiu um grande acréscimo na eficiência do algoritmo. Antes era aplicada a função `resultado(jogo id)` a todos os sucessores de um determinado estado, agora é apenas aplicada aos estados que serão efetivamente visitados, aumentando assim a eficiência pois é evitado realizar operações pesadas como cópias de jogo e atualizações de variáveis de controlo de jogo a estados que nunca serão visitados devido a cortes alfa ou beta.

2.5 Variante 3 do algoritmo minimax

Nesta variante do algoritmo minimax, foi introduzida a técnica das tabelas de transposição, a motivação para a inclusão deste método foi a constatação da existência de jogos semelhantes cuja diferença entre si residia apenas no histórico de jogadas (no caminho). Jogos semelhantes iriam gerar valores de minimax idênticos (e consequentemente jogadas idênticas) caso realizássemos o algoritmo do minimax ao longo da árvore a partir destes. No entanto, este cálculo é dispendioso especialmente quando se tem como objectivo visitar um maior número de nós num tempo limitado. Como tal, considerou-se importante conservar os valores dos minimax.

Para tal, criou-se uma tabela de dispersão por cada árvore cuja chave seria uma estrutura de `jogo-equivalente` e cujo valor na tabela seria o valor do minimax. A descrição dos campos da estrutura e as funções utilizadas encontram-se no final secção 1.1 do relatório. Desta forma, é possível reaproveitar o valor do minimax sem ter que ir visitar nós filhos que iriam devolver informação repetida e aumentar significativamente a profundidade possível de atingir e consequentemente fazer a diferença entre a vitória e a derrota.

Nesta versão foi também aprimorado o algoritmo minimax com cortes alfa e beta face à apresentada na variante 1 e 2. De modo poupar ciclos de processamento, reduziu-se o número de verificações e de variáveis utilizadas, agrupando condições e retirando variáveis redundantes em prol de um algoritmo mais simples e leve que produzisse o mesmo resultado.

3 Funções Avaliação/Heurísticas

3.1 Heurística 1

3.1.1 Motivação

Esta heurística originou do pensamento mais simples possível, o de uma heurística optimista e simples que fosse admissível. A mais óbvia seria a função utilidade como heurística, que faz a diferença de pontos entre ambos os jogadores. No entanto não tinha poder preditivo nenhum, pois só via se naquela jogada tinha mais pontos que o outro. Portanto numa ideia simplista e optimista, seria o nosso jogador admitir que iria ganhar todos pontos disponíveis no tabuleiro para além daqueles que já tinha.

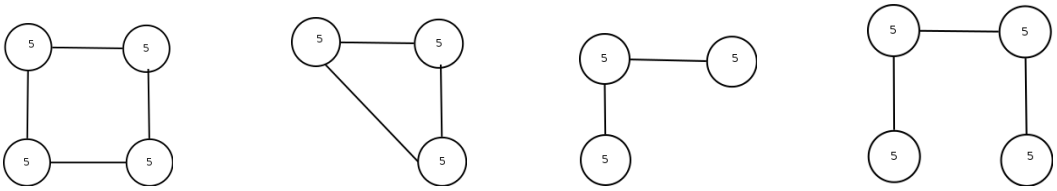
3.1.2 Forma de cálculo

Valor da Heurística1 = **Pontos do Jogador Max** + **Pontos ainda disponíveis** - **Pontos do Jogador Min**.

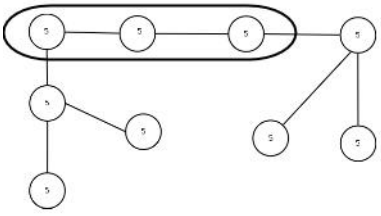
3.2 Heurística 2

3.2.1 Motivação

A segunda heurística foi a primeira heurística com complexidade, esta derivou de denotarmos a importância das cadeias de moedas no jogo. Primeiro tentámos definir o que era uma cadeia, chegando à conclusão que uma cadeia era todas as sequências de moedas ligadas por dois fios terminando ou numa moeda com um único fio ou com mais que dois fios. Ou seja qualquer sequência de moedas do tipo:



Se acabarem como o 1º ou o 4º exemplo, são consideradas cadeias abertas. As cadeias abertas são aquelas em que uma das pontas só tem um fio ligado. Se acabarem como a 2ª ou 3ª cadeia, são consideradas cadeias fechadas. As cadeias fechadas são qualquer qualquer cadeia cíclica, ou seja, que comece e termine na mesma moeda. Por uma questão prática, qualquer sequência de moedas com dois fios ligadas em que ambas as pontas acabe com moedas com mais que dois fios ligados é considerada uma cadeia fechada.



Tendo o que é uma cadeia definida, procedemos a utilizá-las para avaliações heurísticas, sendo que a primeira foi esta, a dois.

3.2.2 Forma de cálculo

A heurística 2 assumia que qualquer cadeia fechada era algo negativo e qualquer cadeia aberta positiva. O objectivo era declarar as cadeias fechadas enquanto uma potencial perda daqueles pontos todos.

Seguindo esta lógica:

Valor da Heurística2 = (Pontos cadeias abertas + pontos jogadorMax) - (pontos cadeias fechadas + pontos jogador oposto)

3.3 Heurística 3

3.3.1 Motivação

A heurística 3 é uma que originou de regras relaxadas, supomos que o jogador alternava sempre entre jogadas. Ou seja, o jogador actual ganhava as moedas com número ímpar de fios ligados e o seguinte as que tinham número par de fios associados.

3.3.2 Forma de cálculo

A heurística 3 calcula-se da seguinte forma:

Valor da Heurística3 = (moedas com nfios ímpares + pontos jogador Max) - (moedas com nfios pares + pontos jogador oposto)

3.4 Heurística 4

3.4.1 Motivação

A heurística 4 é um melhoramento da 2 em que se guarda a cadeia fechada de menor valor e assume-se que o outro jogador a vai consumir porque a decisão é do jogador actual.

3.4.2 Forma de cálculo

A heurística 4 calcula-se da seguinte forma:

Valor da Heurística4 = **(pontos das cadeias abertas + pontos do jogadorMax) - (pontos da cadeia Fechada Menor + pontos jogadorOposto)**

3.5 Heurística 5

3.5.1 Motivação

A heurística 4 verificou-se demasiado simples para alguns tabuleiros/oponentes. Decidimos fazer uma alternativa que não só guardava a fechada de menor valor, como o resto, no entanto duma forma ordenada. Esta cria uma lista de valores de cadeias fechadas em ordem crescente, sendo que o primeiro elemento é a cadeia fechada de menor valor, e o último a cadeia fechada de maior valor. No fim de ter a lista criada somamos as posições ímpares ao adversário e as pares a nós.

Esta assume que o jogador oposto é ótimo.

3.5.2 Forma de cálculo

Valor da Heurística5 = (posições pares lista cadeias fechadas + pontos jogadorMax +pontos cadeias abertas) - (posições ímpares lista cadeias fechadas + pontos jogador oposto)

3.6 Heurística 6

3.6.1 Motivação

A heurística 5 era muito pesada e resultava em decisões erradas, tentámos que apenas tomasse em conta cadeias abertas de maneira ao minimax em si fazer as diferenças de pontos.
Esta acabou por não estar na entrega final porque demonstrou-se como uma heurística muito errada.

3.6.2 Forma de cálculo

Valor da Heurística6= Pontos Jogador Max + Valor das cadeias abertas - Pontos jogador oposto

3.7 Heurística 7

3.7.1 Motivação

Como a heurística 6 não funcionou devidamente e a 5 continuava demonstrar decisões erradas decidimos fazer o oposto, somar apenas as cadeias fechadas. Esta fazia decisões menos incorrectas e foi a escolhida para submissão.

3.7.2 Forma de cálculo

Valor da Heurística7 = Pontos Jogador Max - (Valor das cadeias fechadas + Pontos jogador oposto)

4 Estudo Comparativo

4.1 Estudo variantes do algoritmo minimax/jogador

4.1.1 Critérios a analisar

De forma a comparar as várias variantes previamente apresentadas, foram escolhidos como parâmetros de comparação: o tempo necessário para retornar uma jogada quando não é especificado um tempo limite, o número de nós gerados para um tempo limite e por último a profundidade máxima atingida na árvore de procura por parte dos algoritmos iterativos demonstrados na função `minimax-vbest` quando limitados no tempo.

Só com a comparação destes três parâmetros é possível retirar conclusões quanto á eficiência dos algoritmos, pois um algoritmo mais eficiente não é necessariamente o que analisa mais nós, mas aquele de dentro do tempo permitido analisa o mínimo de nós possível mas chega o mais fundo possível na árvore de procura.

Mas se o objectivo é a profundidade máxima porque ter como factor de análise o número de nós gerados? Porque caso o algoritmo pare na mesma profundidade que outra versão poderemos ter um critério de desempate entre as duas versões testadas.

Para testar as várias versões dos algoritmos, fomos efectuados testes entre a versão mais recente e a anterior, em 4 tabuleiros diferentes com tempo limitado e com tempo ilimitado.

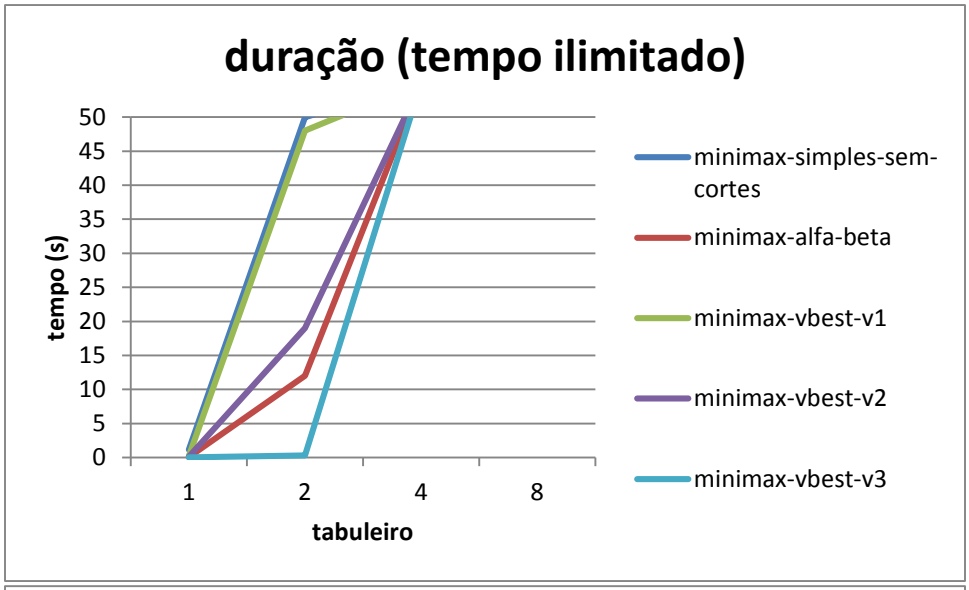
4.1.2 Resultados Obtidos

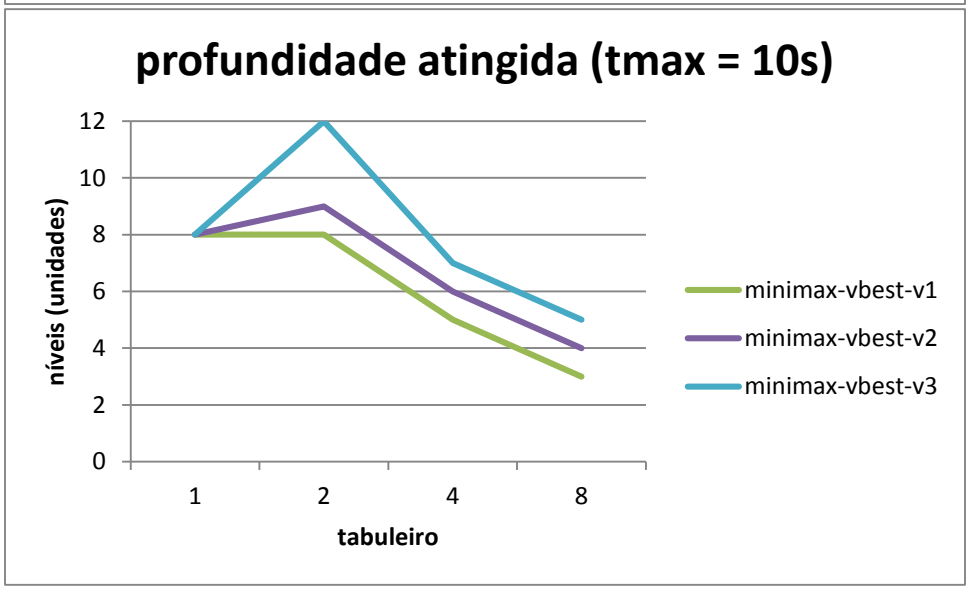
TABULEIRO T1	Duração (tempo ilimitado)	Nós gerados (tmax = 10s)	Profundidade (tmax = 10s)
minimax-simples-sem-cortes	1.15	n/a	n/a
minimax-alfa-beta	0.08	n/a	n/a
minimax-vbest-v1	0.2	9093	8
minimax-vbest-v2	0.13	9093	8
minimax-vbest-v3	0.02	854	8

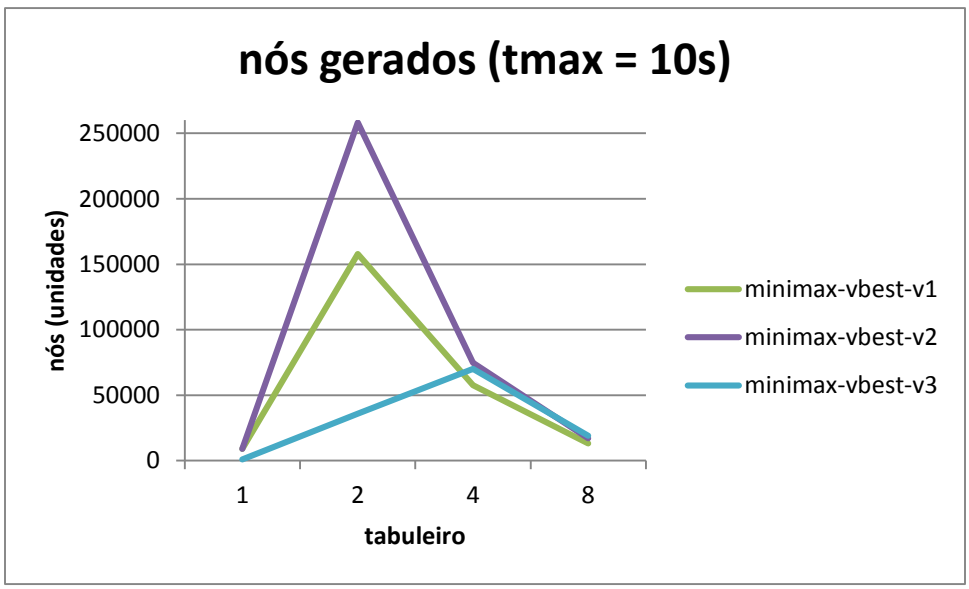
TABULEIRO T2	Duração (tempo ilimitado)	Nós gerados (tmax = 10s)	Profundidade (tmax = 10s)
minimax-simples-sem-cortes	+inf	n/a	n/a
minimax-alfa-beta	12	n/a	n/a
minimax-vbest-v1	48	157.906	9
minimax-vbest-v2	19	258.288	9
minimax-vbest-v3	0.3	35.953	12

TABULEIRO T4	Duração (tempo ilimitado)	Nós gerados (tmax = 10s)	Profundidade (tmax = 10s)
minimax-simples-sem-cortes	+inf	n/a	n/a
minimax-alfa-beta	+inf	n/a	n/a
minimax-vbest-v1	+inf	57.732	6
minimax-vbest-v2	+inf	74.758	6
minimax-vbest-v3	+inf	70.031	7

TABULEIRO T8	Duração (tempo ilimitado)	Nós gerados (tmax = 10s)	Profundidade (tmax = 10s)
minimax-simples-sem-cortes	+inf	n/a	n/a
minimax-alfa-beta	+inf	n/a	n/a
minimax-vbest-v1	+inf	13.044	4
minimax-vbest-v2	+inf	17.099	4
minimax-vbest-v3	+inf	19.114	5


duração (tempo ilimitado)
tempo (s)
50
45
40
35
30
25
20
15
10
5
0
1
2
4
8
tabuleiro
minimax-simples-sem-cortes
minimax-alfa-beta
minimax-vbest-v1
minimax-vbest-v2
minimax-vbest-v3


profundidade atingida (tmax = 10s)
níveis (unidades)
12
10
8
6
4
2
0
1
2
4
8
tabuleiro
minimax-vbest-v1
minimax-vbest-v2
minimax-vbest-v3


nós gerados (tmax = 10s)
nós (unidades)
250000
200000
150000
100000
50000
0
1
2
4
8
tabuleiro
minimax-vbest-v1
minimax-vbest-v2
minimax-vbest-v3

4.1.3 Comparação dos Resultados Obtidos

Em relação ao tempo de execução do algoritmo (útil quando chegamos ao fim da árvore) podemos concluir que as várias versões minimax foram sempre melhorando neste aspecto. Existe um aumento significativo no desempenho com a introdução dos cortes alfa-beta (`minimax-alfa-beta`), um aumento considerável com a aplicação da jogada apenas quando os nós não são cortados (`minimax-vbest-v2`) vs. quando aplicávamos a todos sem pensar que poderiam vir a ser cortados (`minimax-vbest-v1`). Após estes melhoramentos, conseguimos um aumento brutal de desempenho com a introdução das tabelas de transposição, que minimizaram o numero de nós visitados permitindo uma busca mais rápida chegando assim a níveis mais fundos de profundidade na árvore de procura (`minimax-vbest-v3`).

Os outros dois factores de comparação têm de ser analisados como um conjunto, pois da versão 1 para a versão 2 (vbest) foi conseguido, para o mesmo tempo, um aumento do numero de nós visitados. Com a versão 3 esse numero foi reduzido, não porque se perdeu desempenho, mas porque com as tabelas de transposição o valor do minimax não teve de ser calculado em cada nó, mas sim reaproveitado dos valores já guardados. Com esta tabela conseguimos reduzir o numero de nós visitados e assim aumentar a profundidade atingida na árvore de procura, o que indica um melhoramento substancial sobre a versão 2.

Concluímos assim, que a `minimax-vbest-v3` é a melhor versão de minimax desenvolvida, pois conta com os cortes alfa-beta da primeira entrega, as optimizações de geração introduzidas na versão 2, e também com a introdução das tabelas de transposição que permitem o aproveitamento de valores do minimax préviamente calculados.

4.2 Estudo funções de avaliação/heurísticas

Ao longo do desenvolvimento do projecto foram desenvolvidas várias heurísticas. Infelizmente nem todas tinham o desempenho que era desejado, comportando-se de uma maneira errada em alguns casos. Isto exceptuando a função utilidade que pode, e foi, usada como heurística.

4.2.1 Critérios a analisar

De modo a analisar qual seria a heurística melhor decidimos usar como critérios a partir de que profundidade a heurística converge numa resposta, o número de nós gerados e a profundidade máxima que o minimax conseguiu chegar. Considerámos que, como a heurística é usada apenas se o tempo acabar, o tempo não é um critério adequado para esta análise.

4.2.2 Testes Efectuados

O teste utilizado foi a utilização mais usual de uma heurística, foi a utilização da mesma para a decisão de uma jogada através do jogador minimax-vbest em vários tabuleiros. Este é a melhor maneira de verificar quão útil pode ser a heurística num caso de jogo.

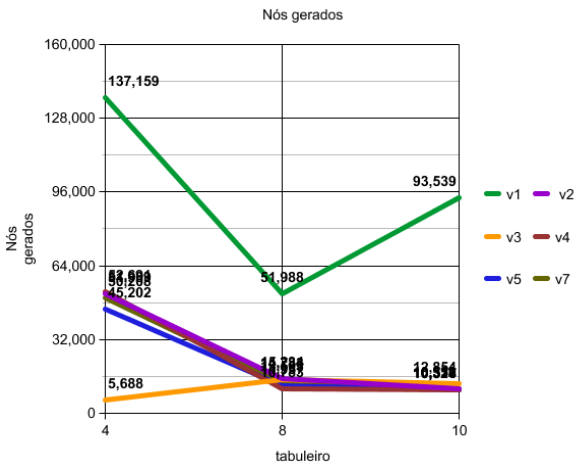
4.2.3 Resultados Obtidos

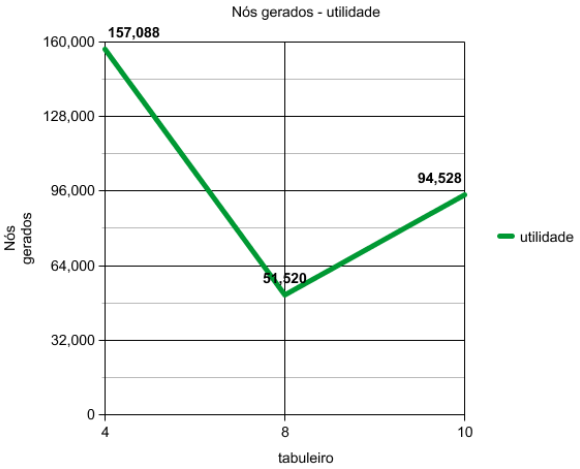
Tempo = 7s

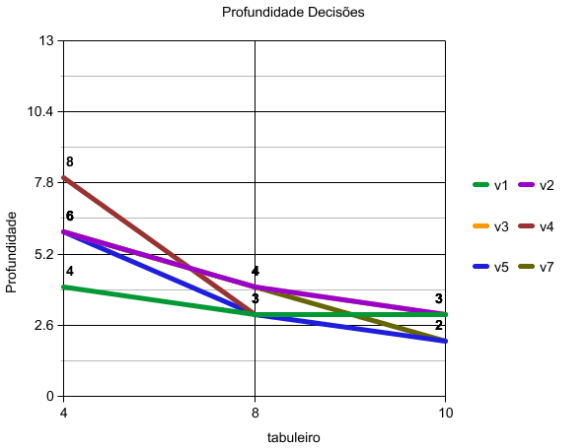
Tabuleiro t4	Nós Gerados	Profundidade Máxima	Profundidade Decisão
v1	137159	12	4
v2	51909	7	6
v3	58688	7	6
v4	52691	9	8
v5	45202	7	6
v7	50208	7	6
utilidade	157088	12	4

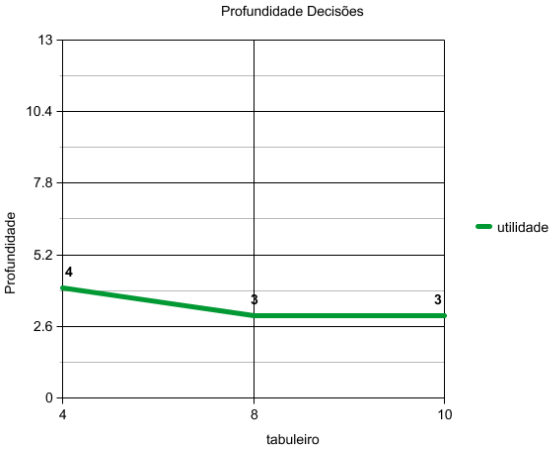
Tabuleiro t8	Nós Gerados	Profundidade Máxima	Profundidade Decisão
v1	51988	6	3
v2	15234	5	4
v3	14783	5	4
v4	10783	4	3
v5	11957	4	3
v7	13503	5	4
utilidade	51520	6	3

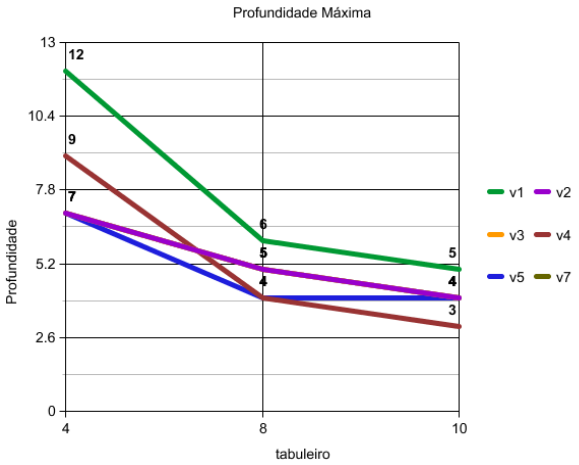
Tabuleiro t10	Nós Gerados	Profundidade Máxima	Profundidade Decisão
v1	93539	5	3
v2	10559	4	3
v3	12854	4	3
v4	10370	4	3
v5	10329	4	2
v7	10512	4	2
utilidade	94528	5	3

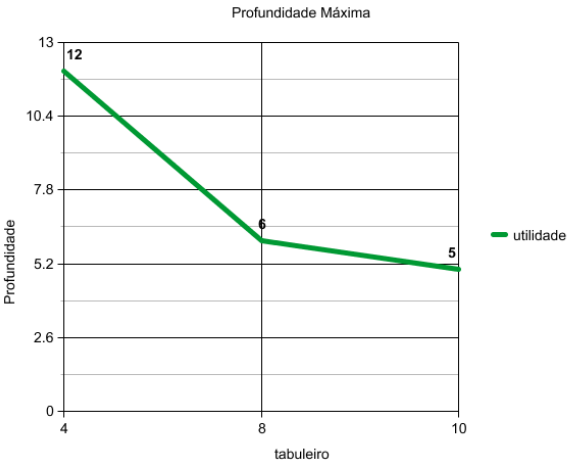

Nós gerados
160,000
128,000
96,000
64,000
32,000
0
Nós gerados
137,159
93,539
51,988
5,688
4
8
10
tabuleiro
v1
v2
v3
v4
v5
v7


Nós gerados - utilidade
160,000
128,000
96,000
64,000
32,000
0
Nós gerados
157,088
94,528
51,520
4
8
10
tabuleiro
utilidade


Profundidade Decisões
13
10.4
7.8
5.2
2.6
0
Profundidade
8
6
4
4
3
3
2
4
8
10
tabuleiro
v1
v2
v3
v4
v5
v7


Profundidade Decisões
13
10.4
7.8
5.2
2.6
0
Profundidade
4
3
3
4
8
10
tabuleiro
utilidade


Profundidade Máxima
13
10.4
7.8
5.2
2.6
0
Profundidade
12
9
7
6
5
5
4
4
3
4
8
10
tabuleiro
v1
v2
v3
v4
v5
v7

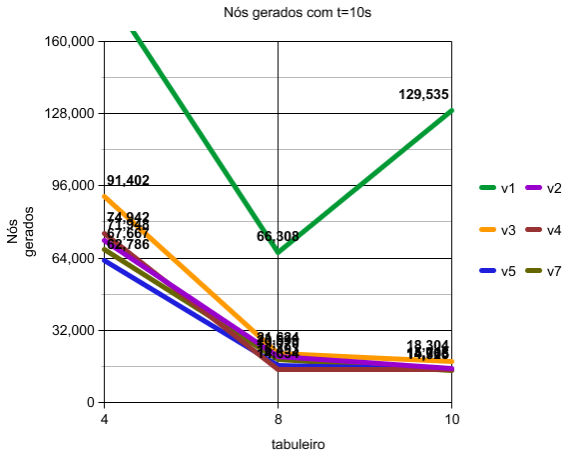

Profundidade Máxima
13
10.4
7.8
5.2
2.6
0
Profundidade
12
6
5
4
8
10
tabuleiro
utilidade

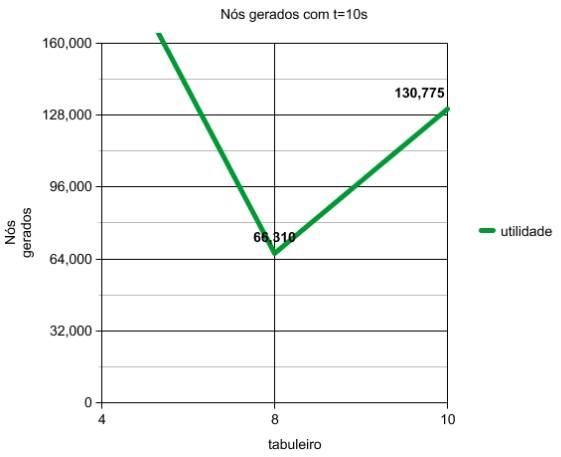
Tempo = 10s

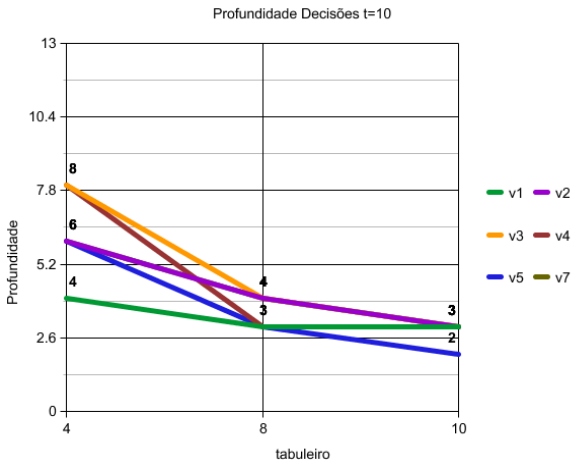
Tabuleiro t4	Nós Gerados	Profundidade Máxima	Profundidade Decisão
v1	183891	12	4
v2	71948	7	6
v3	9142	9	8
v4	74942	9	8
v5	62786	7	6
v7	67667	7	6
utilidade	212119	12	4

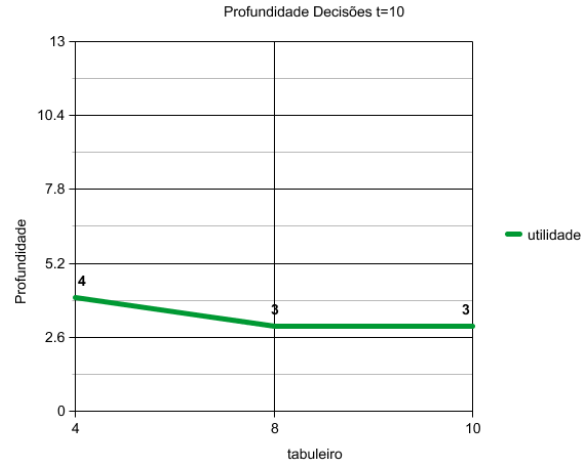
Tabuleiro t8	Nós Gerados	Profundidade Máxima	Profundidade Decisão
v1	66308	6	3
v2	20599	5	4
v3	21624	5	4
v4	14634	4	3
v5	16493	4	3
v7	19070	5	4
utilidade	66310	6	3

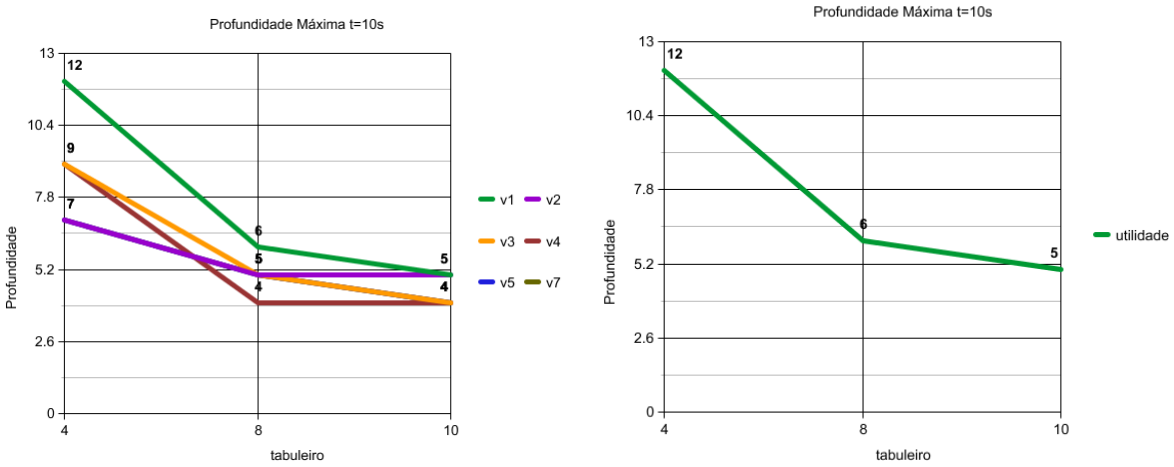
Tabuleiro t10	Nós Gerados	Profundidade Máxima	Profundidade Decisão
v1	129535	5	3
v2	15018	5	3
v3	18304	4	3
v4	14703	4	3
v5	14766	4	2
v7	14325	4	3
utilidade	130775	5	3











4.2.4 Comparação dos Resultados Obtidos

A heurística 1 aproxima-se muito da utilidade pela sua simplicidade explorando muitos mais nós pela mesma razão. Todas as heurísticas baseadas em cadeias tiveram um comportamento semelhante, sendo que devido à ordenação de valores a v5 está especialmente pesada. A v3 está também para trás no número de nós gerados devido à procura de todos os fios de todas as moedas.

4.3 Escolha do jogador-minimax-vbest

Com base nos resultados obtidos nas duas comparações anteriores, devem descrever e justificar quais as variantes escolhidas para implementar o jogador jogador-minimax-vbest, bem como a função de avaliação utilizada para este jogador.

A escolha do do jogador-minimax-v-best foi complicada, visto que haviam heurísticas de desempenho semelhante, portanto tentámos excluir as heurísticas que davam respostas claramente erradas.

A heurística 1 era demasiado optimista não trazendo nenhuma vantagem em relação à utilidade. A heurística 2, apesar de fazer uma análise correcta das cadeias existentes no ecrã não atribuía de maneira realista os pontos resultantes das mesmas, resultando em respostas erradas. A heurística 3 foi imediatamente posta de lado pois a ideia das regras relaxadas, em que o jogador alternava sempre, tornava as sequências de jogadas consecutivas do mesmo jogador menos importantes do que eram resultando em respostas erradas. Entre as heurísticas 4, 5 e 7 é que surgiram as verdadeiras dúvidas. Como todas usavam identificação de cadeias e tinham vantagens e desvantagens diferentes, foi necessário uma análise mais profunda.

Ultimamente, como a 4 e a 5 se provaram erradas nas decisões entre duas jogadas ao serem testadas contra a utilidade, ganhando apenas com uma diferença mínima, decidimos usar a 7 para como função de avaliação para o `jogador-minimax-vbest`.

Por último, tal como abordado na secção 2.5 onde descrevemos as variantes do algoritmo minimax implementadas, foi escolhido a versão 3 do minimax com cortes alfa-beta e tabela de transposição, pois tal como evidenciado através de vários testes práticos descritos na secção 4.1.2, foi este que revelou uma *perfomance* superior e como tal, era esta que conseguia chegar mais longe na árvore de procura e conseguia gerar mais nós juntamente com a função de avaliação, e por consequência era o jogador que tinha mais probabilidade de levar-nos à vitória.