Grupo Nº 39

TÉCNICO LISBOA

Inteligência Artificial

1.º Semestre 2015/2016

IA-Tetris

Relatório de Projeto

Índice

1 Implementação Tipo Tabuleiro e Funções do problema de Procura

1.1 Tipo Abstrato de Informação Tabuleiro

O tipo tabuleiro foi implementado com uma matriz de dimensões 18x10. Esta matriz contém apenas valores booleanos True e NIL. Inicialmente a matriz é inicializada a NIL e quando uma peça é colocada no tabuleiro, as posições ocupadas tomam o valor True.

Decidimos usar esta estrutura pois as operações mais comuns tais como preencher uma posição (tabuleiro-preenche!) e saber se uma posição está preenchida (tabuleiro-preenchido-p) apenas requerem um acesso à memória, ou seja constante, o que torna esta opção viável.

Nas considerações feitas acerca desta estrutura pensámos em implementar uma matriz 19x10 em que a 19ª linha continha a altura máxima de cada coluna e, com isto a função tabuleiro-altura-coluna seria também constante. No entanto acabámos por não implementar esta via pois o número de operações aritméticas na função tabuleiro-preenche! iria aumentar significativamente.

Em termos de memória a representação escolhida não é a melhor, mas também não é péssima. Visto que a nossa maior preocupação era em termos de tempo, isto não foi um fator preocupante.

1.2 Implementação de funções do problema de procura

Função accoes

Esta função recebe um estado e verifica se o tabuleiro nesse estado está ou não preenchido.
Caso não esteja preenchido, é verificada a próxima peça a colocar com todas as posições possíveis e devolvida a lista de accoes possiveis.

Função resultado

Esta função apenas aplica a ação recebida ao estado recebido, ou seja coloca a peça no tabuleiro. Se o resultado desta função ultrapassar o topo do tabuleiro devolve-se o estado sem remover linhas, caso contrário removem-se as linhas (se existirem), calculam-se os pontos obtidos e devolve-se o estado.

2 Implementação Algoritmos de Procura

2.1 Procura-pp

O algoritmo de procura em profundidade primeiro foi implementado de uma maneira bastante simples. A estrutura que decidimos usar para representar um nó na árvore de procura foi uma lista com três elementos, em que o primeiro corresponde ao estado, o segundo corresponde ao nó pai e o terceiro representa a ação tomada para chegar do nó pai ao nó atual. Os nós são colocados numa estrutura FIFO à medida que são gerados e portanto o último a ser gerado vai ser o primeiro a ser expandido. Assim que é encontrada uma solução, o algoritmo sai do ciclo principal e faz agora a construção do caminho.

A construção do caminho começa no nó objetivo e vai regredindo na árvore até encontrar o nó raiz enquanto se colocam as ações tomadas na lista de ações a retornar.

2.2 Procura-A*

Implementamos o algoritmo a* de uma maneira relativamente simples. O ciclo principal escolhe o nó com menor valor de f(n) para expansão e gera todos os seus sucessores colocando-os na fila de prioridade. Quando o nó expandido é objectivo o algoritmo termina e é agora calculado o caminho de uma maneira semelhante à que foi usada na procura-pp. O nó é uma estrutura que permite facilmente calcular o caminho até à raiz como está descrito no segmento abaixo.

- **Estruturas usadas**

O conjunto de nós abertos é um conjunto sobre o qual apenas duas ações são requeridas, retirar o nó com menor valor e adicionar um novo nó. Tendo isto em conta, é óbvio que uma estrutura que mantenha algum sentido de ordenação em relação ao valor terá uma melhor eficiência do que uma que percorra a lista inteira sempre que se faz uma destas ações. Ao analisarmos o algoritmo A* apercebemo-nos que as inserções são muito mais comuns que as remoções (cerca de 20 a 30 vezes no caso do Tetris) e portanto optámos por usar um *binary heap*. O *binary heap* tem um custo médio de inserção constante, O(1) e custo de remoção logarítmico, O(log(n)), e portanto resolve o problema eficientemente.

A nossa implementação do *heap* tem em conta que a função f(n) pode ser um pouco cara de computar e portanto calcula-a apenas uma vez e guarda-a numa estrutura juntamente com o nó. Também garantimos que há uma completa abstração e portanto este *heap* poderia ser usado para problemas futuros. A estrutura usada para implementar o *heap* é um *array* de tamanho dinâmico, ou seja, sempre que o *heap* esgota a capacidade do *array*, este é realocado para o dobro do tamanho.

O conjunto de nós fechados, visto que implementámos a versão *tree-search* do A*, é apenas relevante a ação de adicionar um elemento. Visto isto, escolhemos usar uma lista ligada em que a inserção é feita em tempo constante.

Cada nó é representado como uma estrutura em *lisp* que contém o estado, o índice de expansão (portanto o índice na lista de nós explorados), o índice do nó pai e acção que foi tomada para gerar este nó. Assim é possível fazer a construção do caminho após a terminação do algoritmo sem usar muita memória de forma semelhante aquela que usamos na procura-pp.

- **Procura A* limitada**

Ao testarmos a procura A* com o problema do Tetris apercebemo-nos que seria muito complicado de encontrar uma heurística que reduzisse o valor médio de ramificação para um valor aceitável e que portanto poderia terminar em tempo útil. A nossa primeira otimização foi simplesmente limitar o número de nós gerados. Assim que atingisse o limite, o algoritmo retornaria o último nó expandido (com maior f(n)). É óbvio que assim perdemos bastante qualidade mas pelo menos conseguiríamos ver algum resultado e prosseguir com mais otimizações.

- **Procura A* parcial com previsão limitada (Procura Previsão)**

Para melhorar o algoritmo, fizemos uma alteração que permitia acabar em tempo útil e chegar a um nó terminal. Para isto, mudámos o comportamento do A* para que assim que esgotasse o limite de nós, começava de novo o algoritmo mas começando do segundo nó do melhor caminho encontrado. Fazendo isto iterativamente até chegar a um nó terminal conseguimos chegar a um algoritmo que usa memória constante e a complexidade temporal é linear com o número de peças colocadas (profundidade da solução). É óbvio que isto resulta numa perda de otimalidade e, em geral, de qualidade.

- **Procura A* parcial com previsão limitada por nível de profundidade (Procura PreNivel)**

Ao observarmos o comportamento desta modificação do A*, apercebemo-nos que havia muito desperdício ao recomeçar o algoritmo a cada, no nosso caso, 10 mil nós. Perdia-se toda a informação dos nós já expandidos e gerados sempre que se recomeçava, e foi no sentido de eliminar esta ineficiência que pensamos na próxima modificação. O resultado foi um algoritmo simples: corre-se o algoritmo a* e quando é atingido um limite de 10000 nós, são removidos todos os nós gerados do nível mais acima que ainda não foi removido. Faz-se isto até chegar a uma solução. A memória máxima continua a ser constante com a profundidade e a ramificação (depende do limite definido) e a complexidade temporal é linear. A qualidade da solução retornada é que vai depender fortemente do limite definido, da ramificação, da profundidade e da heurística.

Para tornar isto eficiente tivemos que usar uma nova estrutura para representar a fronteira, que é apenas um *array* de *heaps*. A operação de *push* continua a ser igualmente eficiente (visto que tem que se descriminar a qual *heap* é que se pretende inserir o elemento) e a operação de *pop* também continua a ser bastante eficiente desde que o *array* seja pequeno (corresponde apenas a procurar o mínimo entre todos os *heaps*).

3 Funções Heurísticas

3.1 Heurística Buracos

3.1.1 Motivação

- Um tabuleiro limpo (sem buracos) permite fazer linhas com mais facilidade, criando uma maior probabilidade de fazer mais pontos com uma só peça. Ao mesmo tempo, um buraco presente numa linha impede-a de ser feita com apenas uma peça, para tal, é primeiro necessário que as linhas que estão por cima dessa sejam completas e só depois e possível fazer a linha com o buraco. Este problema novamente limita a possibilidade de máximizar a pontuação por peça.
- Para esta heurística, o tabuleiro presente no estado é suficiente para ser calculada.

3.1.1 Forma de Cálculo

Esta heurística é simples não só no seu conceito, mas também na sua implementação. Na teoria, um buraco ocorre quando existe uma posição não preenchida e existe uma posição acima desta que se encontra preenchida, criando a impossibilidade de fazer essa linha sem primeiro completar as linhas que se encontram acima. Na figura 1, verificam-se três buracos, nas linhas 2 e 4. Estes buracos dificultam não só o preenchimento dessas linhas mas também de maximizar a pontuação possível com uma só peça. Caso uma peça I aparece-se, em vez de fazer 500 pontos, apenas poderá fazer 300 (linhas 1 e 3).

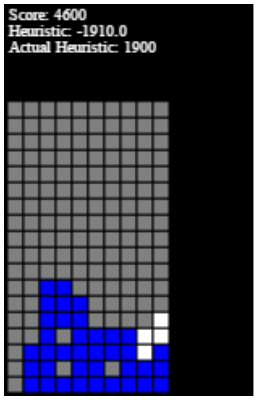

Figura 1: Exemplo de tabuleiro com buracos.

A sua implementação é bastante direta. O algoritmo vai à procura de uma posição não preenchida, quando tal ocorre, verifica se a posição acima se encontra preenchida. Caso essa situação se verifique, o algoritmo vê quantas posições estão vazias para baixo dessa e incrementa a variável 'buracos'. Essa é a variável devolvida pela função, depois de multiplicada por um escalar para lhe dar relevância.

3.2 Heurística Altura Tabuleiro Máxima

3.2.1 Motivação

- Quanto mais alto estiver o tabuleiro, mais perto está de perder. Esta heurística permite prevenir que o tabuleiro fique muito alto e ao mesmo tempo incentiva o agente a preencher linhas, pois quando tal acontece, a altura diminui.

- Para esta heurística, o tabuleiro presente no estado é suficiente para ser calculada.

- Foram pensadas várias vertentes desta heurística, nomeadamente, em vez da altura máxima, foi considerada a possibilidade de a heurística tomar em consideração a altura média ou que o valor da heurística variasse não linearmente, mas quadraticamente, ganhando maior valor quanto maior fosse a altura. A altura de tabuleiro média foi descartada devido a o facto de que não oferecer uma visão realista do tabuleiro, pois não toma em consideração outros valores com uma dispersão à média maior. A altura sofrer uma variação quadrática foi algo fortemente discutido devido aos benefícios inerentes. Quando o tabuleiro tem uma altura baixa, não é muito relevante tentar diminuir o valor da heurística, pois este não necessita de tentar completar linhas baixando a altura. No decorrer do jogo, quanto maior for a altura, mais imperativo se torna converter linhas em pontos. No entanto, após os resultados do algoritmo genético, verificámos que a altura máxima produzia melhores resultados.

3.2.1 Forma de Cálculo

Esta heurística, tal como a anterior, é bastante simples na sua intenção e implementação.

As colunas são percorridas uma a uma, caso a altura de uma das colunas seja maior do que as anteriores, a variável 'máximo' é atualizada com o valor da nova coluna. Este valor e calculado através da função tabuleiro-altura-coluna. No fim do ciclo é devolvida essa variável.

Na figura 2 existem várias colunas com altura distintas. No entanto, esta heurística apenas se interessa com a altura máxima das colunas do tabuleiro. Como se verifica, a coluna 8 possui a posição com a altura mais elevada, 13. Este é o valor devolvido pela heurística.

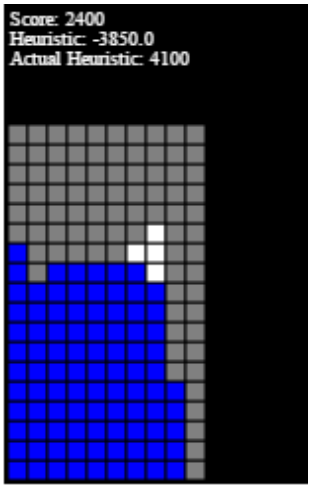

Figura 2: Exemplo de tabuleiro com altura máxima.

3.3 Heurística Linhas parciais

3.3.1 Motivação

- Linhas quase completas são vantajosas para maximizar os pontos, desde que não estejam bloqueadas por outras linhas e, portanto esta heuristica diz que se houverem linhas muito preenchidas é provável que se consiga uma melhor pontuação
- Para esta heuristica é usado o tabuleiro e as peças por colocar
- No início tinhamos uma forma simples de calcular o número de linhas bloqueadas. Apercebemo-nos que isto não era tão eficaz como esperado, e atribuímos 0 pontos se as linhas estivessem bloqueadas.

3.3.1 Forma de Cálculo

Para calcular linhas parciais, é percorrido todo o tabuleiro. Para cada linha vê-se quantos blocos estão preenchidos e, caso esteja vazio, vê-se se está bloqueado por cima. Caso esteja vazia e bloqueada, é atribuido 0 pontos à linha, caso esteja preenchido é incrementada uma variavel que índica o número de blocos preenchidos. Quando a linha foi completamente percorrida, é atribuida uma pontuação dependendo do número de blocos preenchidos segundo a tabela 1. Assim valorizam-se linhas que apenas têm 1 bloco não preenchido, porque é mais facil fazer mais que uma linha por peça. Caso os blocos não estejam seguidos, é dada uma penalização. Também é tido em conta o número de peças que se usam a encher cada linha, como se pode observar na tabela, decrementando o total de peças pelo valor indicado. Quando não há mais peças por colocar o algoritmo termina.

Blocos preenchidos	Pontuação dada	Peças gastas
5	75	1.25
6	100	1
7	150	0.75
8	187.5	0.5
9	225	0.25

Tabela 1: Pontuação atribuída e peças usadas em função dos blocos preenchidos

Na figura 3 podemos observar várias linhas quase completas, por exemplo, a de baixo será atribuido uma pontuaçao de 187.5. A segunda linha tem 4 blocos, mas como estão separados apenas será atribuido uma pontuação de 75. As restantes linhas contribuem 0 para o cálculo da heuristica e o número de peças restantes também não é determinante para este caso.

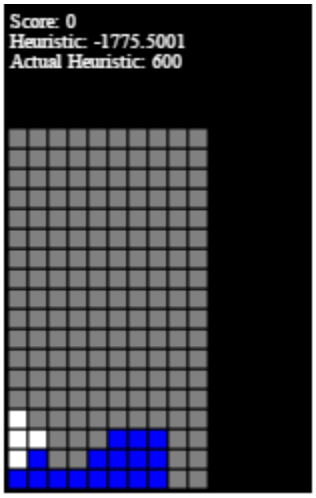

Figura 3: Exemplo de tabuleiro com linhas incompletas.

3.4 Heurística Máximo-Pontos-Estado

3.4.1 Motivação

- Esta heurística permite estimar os pontos que são possiveis obter tento em conta não só o tabuleiro atual mas também as peças futuras. É obtida através de uma relaxação de várias restrições, isto é, assume-se que as peças podem ser decompostas em blocos mais pequenos e colocadas em qualquer sitio no tabuleiro. Também se assume que todos os blocos do tabuleiro podem ser reorganizados de forma a maximizar os pontos.
- Para a heurística, foi utilizado o tabuleiro e as pecas por colocar.
- A nossa ideia foi criar uma heuristica simples que nos desse um *upper bound* para os pontos possivelmente obtidos no futuro

3.4.1 Forma de Cálculo

Em primeiro lugar, esta heurística percorre o tabuleiro determinando o número de posições preenchidas, somando a esse valor o número de posições que irão ser preenchidas com as próximas peças. De seguida, a variável 'total_linhas' recebe o número de linhas que é possivel fazer no melhor dos casos com o número de posições que irão ser preenchidas. No resto do algoritmo, a variavel que irá ser devolvida, 'total_score', vai aumentando tendo em conta o número máximo possível de pontos. Em primeiro verifica-se quantas linhas e possivel fazer fazendo a divisão por quatro e multiplicando pelas pontuação correspondente a fazer quatro linhas numa só jogada. Seguidamente, a variável 'total_linhas' é atualizada para conter o novo número de linhas depois das que foram usadas anteriormente. Este processo e repetido até tentar fazer uma linha. Terminadas essas instruções, é devolvida a variável 'total_score', que contém o valor da soma do processo descrito anteriormente.

Tendo em conta que esta heuristica prevê os pontos de uma relaxação das restrições do problema, então podemos dizer que é uma previsão optimista para o problema do tetris, ou seja, uma heuristica admissivel. Contudo este cálculo apenas tem em conta as peças e o número de blocos no tabuleiro e portanto dá relativamente pouca informação sobre tabuleiros que estão a um mesmo nível de profundidade, o que se relevou pouco útil na prática.

3.5 Heurística Bumpiness

3.5.1 Motivação

- Um tabuleiro com uma grande variação na oscilação é em geral um tabuleiro mais difícil de completar e, consequentemente, mais difícil fazer pontos. Esta heurística penaliza os casos em que a variação á muito elevada de forma a evita-lo.
- Para esta heurística, o tabuleiro presente no estado é suficiente para ser calculada.
- Pouco foi mudado nesta heurística após a sua implementação, apenas foi estudado o peso para dar relevância, no entanto chegamos a conclusão que esta heurística não influenciava o jogo e acabamos por não a usar.

3.5.1 Forma de Cálculo

Mais uma vez, a heurística não é muito complexa na sua implementação. As colunas são percorridas e a altura da coluna é calculada com o auxílio da função tabuleiro-altura-coluna. O valor da coluna anterior é sempre guardado e depois ser subtraído á altura da coluna em análise. O valor devolvido é a soma de todas as variações.

Na figura 4, verifica-se uma diferença considerável entre as alturas das colunas. Entre a primeira e a segunda coluna, não existe variação, logo o valor da variável 'total' é 0. Nas colunas que se seguem, nota-se uma diferenca de uma linha entre as colunas 2 e 3, e 3 e 4, mudando o valor da variável para 2. Na coluna 5, existe uma grande diferença em relação à anterior, aumentando 'total' em 3 unidades. Este processo e repetido ate percorre todas as colunas.

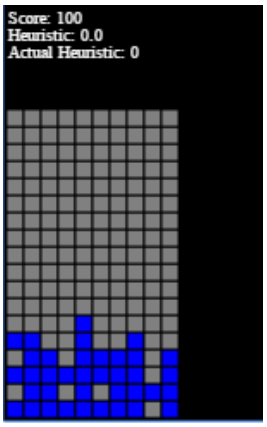

Figura 4: Exemplo de tabuleiro com bumpiness.

4 Estudo Comparativo

Nas seguintes secções, é importante notar que os coeficientes usados para dar pesos ás diferentes heurísticas foram selecionados por uma implementação de um algoritmo genético que será descrito posteriormente na secção 5.

4.1 Estudo Algoritmos de Procura

4.1.1 Critérios a analisar

Para comparar os algoritmos de procura, usámos os critérios de memória, tempo de execução e qualidade das jogadas. Estes critérios mostraram-se suficientes e úteis para avaliar as procuras de forma a optar pela melhor.

4.1.2 Testes Efetuados

Para testar os algoritmos em termos de memória, efetuámos 20 testes para cada número de peças (considerámos de 2 a 10 peças) e calculámos os nós expandidos e os nós gerados.

Estes testes serviram também para calcular o tempo de execução, pois este tempo é linear em relação aos numero total de nós.

Para testar a qualidade das jogadas, observámos o número de pontos total.

Foram feitos 10 jogos para cada nível de peças e calculado médias, com um tabuleiro aleatório usando a função dada no *utils.lisp* com argumentos 1.0 e decaimento de 0.1. As peças também foram calculadas aleatoriamente.

4.1.3 Resultados Obtidos

- **Teste 1:** Comparação de procuras usando heurística pessimista otimizada para A*.

  Heuristica usada: 78 * altura maxima – 0.46* maximo pontos + 0.25 * linhas parciais

  Conforme se verifica na figura 5, existe uma grande similaridade entre o número de nós gerados para todos os algoritmos de procura. Nota-se também que o número de nós expandidos é exatamente o mesmo. Isto significa que, em termos de memória e de tempo, os algoritmos são equivalentes (para 10 peças).

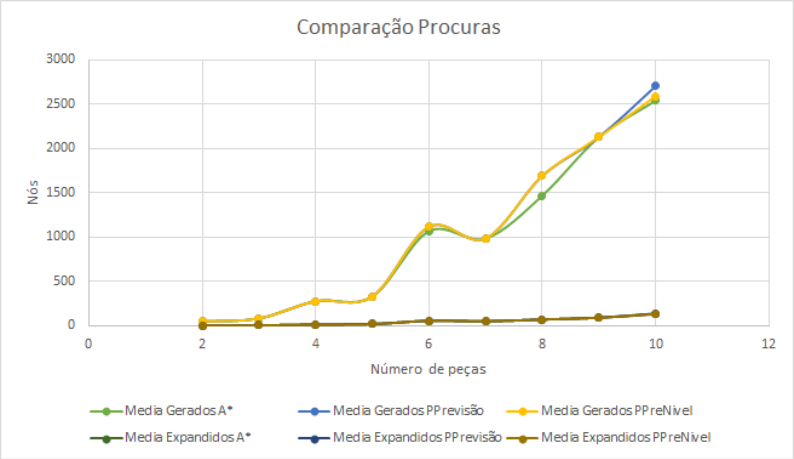

Figura 5: Comparação da Procura A em relação às outras procuras,*

Na figura 6, pode verificar-se que todas as procuras são equivalentes, isto acontece porque, até 10,000 nós, não há diferenças na execução dos algoritmos, logo os pontos gerados são os mesmos.

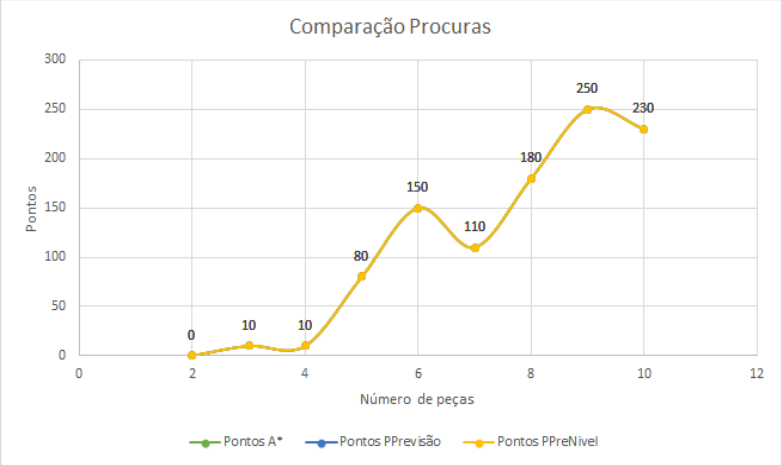

Figura 6: Comparação da Procura A em relação às outras procuras, analisando o número de pontos.*

Os resultados não são muito satisfatórios porque a heurística foi uma heuristica que premitisse o A* acabar antes de antigir os 10,000 nós, resultando numa procura rápida e pessimista.

- **Teste 2:** Procura PreNivel, heurística mais otimista que a anterior.

  Heuristica usada: 11* altura maxima + 0.1 * linhas parciais + 0.66 * maximo pontos + 5.6 * buracos

  Na figura 7 nota-se uma diferença entre o número de nós gerados da procura Pprevisão e as restantes. Esta diferença acontece porque a Procura A* está limitada a 20.000 nós e a procura PreNivel está também limitada apesar de continuar a execução após o limite eliminando os primeiros níveis de profundidade. Com isto melhorámos os níveis de memória e tempo de execução das procuras A* e PreNivel.

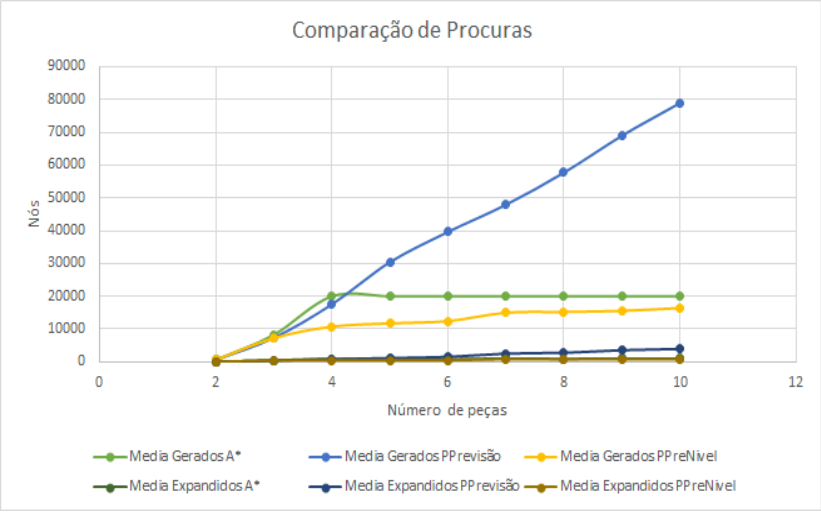

Figura 7: Comparação da Procura PreNivel em relação às outras procuras,

Observando a figura 8, é evidente que a procura PreNivel é melhor no que toca à qualidade das jogadas. Isto acontece devido às otimizações dessa procura explicadas na secção 2.2.

Como se verifica, o número de pontos da procura PreNivel é em geral para esta heuristica, maior do que as outras procuras.

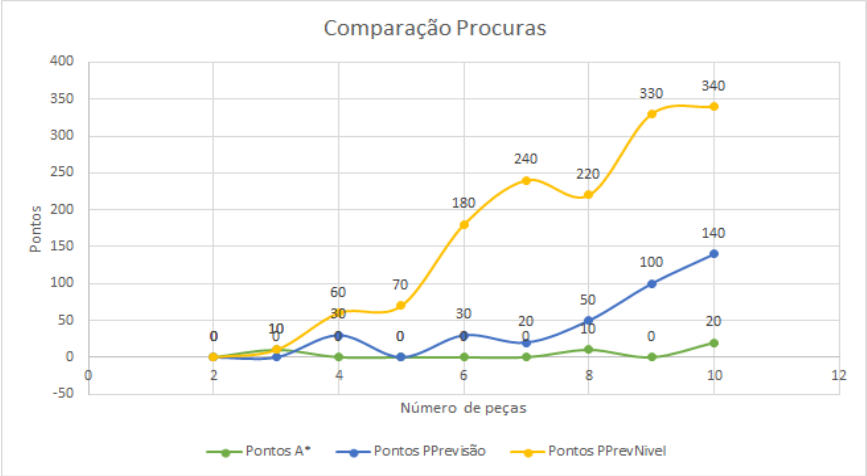

Figura 8: Comparação da Procura PreNivel em relação às outras procuras, analisando o número de pontos.

- **Teste 3:**Procura em Profundidade Primeiro, heurística otimizada para A*.

Finalmente, ao comparar o número de nós gerados na figura 9, verificámos que a Procura em Profundidade Primeiro ocupa menos memória e demora menos tempo que as restantes.

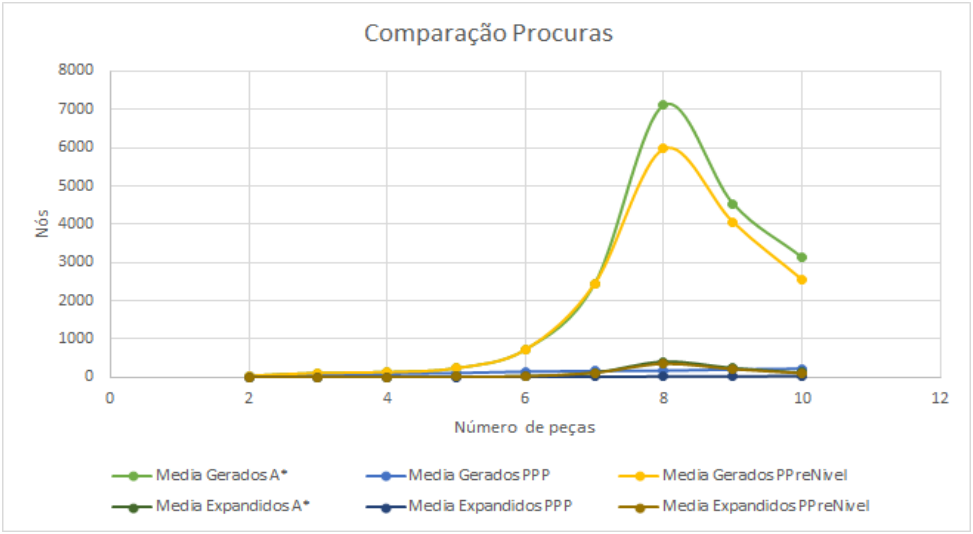

Figura 9: Comparação da PPP em relação às outras procuras, analisando o número de nós.

No entanto, como se vê na figura 10, no que toca à qualidade das jogadas, a Procura em Profundidade Primeiro é considerávelmente pior que as outras, estando novamente a Procura PreNivel com vantagem.

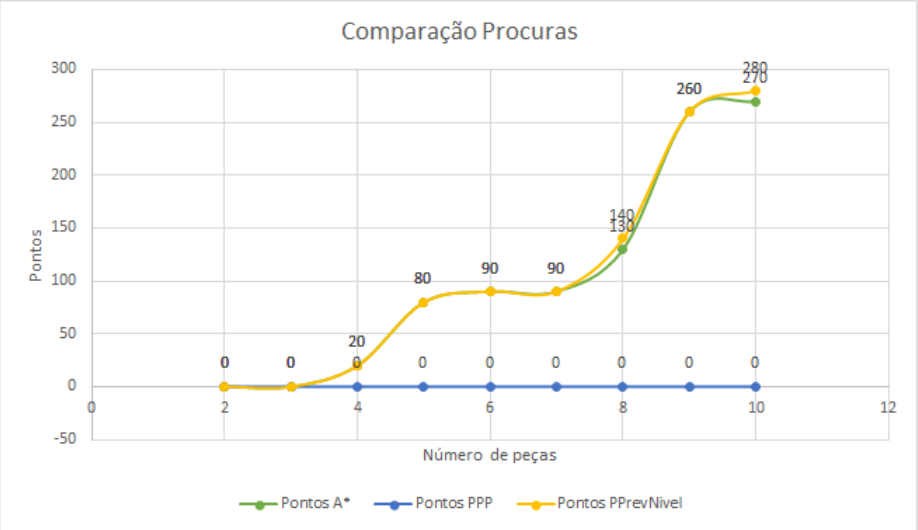

Figura 10: Comparação entre a PPP em relação às outras procuras, analisando o número

4.1.4 Comparação dos Resultados Obtidos

Após a análise dos resultados obtidos na secção 4.1.3. é possível concluir que na maioria dos casos, a Procura PreNivel é superior em relação à qualidade das jogadas fazendo uma gestão aceitável da memória. Consequentemente, foi essa que decidimos usar ao longo do projeto.

Dado que a Procura em Profundidade Primeiro é não é a melhor em relação à qualidade das jogadas, foi descartada pois esse é o nosso principal critério.

4.2 Estudo funções de custo/heurísticas

4.2.1 Critérios a analisar

Para avaliar as heurísticas, definimos como principal critério o número de pontos que cada heurística consegue atingir para um mesmo tabuleiro, ou seja a qualidade das jogadas. O algoritmo de procura utilizado foi o Procura PreNivel, pois foi o selecionado pelo grupo para algoritmo final. A função de custo-caminho foi sempre a mesma, que corresponde à qualidade descrita no enunciado (pontos x -1).

4.2.2 Testes Efetuados

Para comparar as heurísticas, testamos cada uma individualmente (mas juntamente com a maximo-pontos-estado) bem como a junção de algumas as heurísticas que foram a nossa heurística final.

Foram feitos 10 jogos para cada nível de peças e calculado médias, com um tabuleiro aleatório usando a função dada no *utils.lisp* com argumentos 1.0 e decaimento de 0.1. As peças também foram calculadas aleatoriamente.

4.2.3 Resultados Obtidos

Nesta figura 11 foi utilizado um tabuleiro parcialmente preenchido para testar as heurísticas. Como se verifica, a heurística final é a que maximiza o número de pontos. Conseguimos observar algum sentido de relevância para as restantes heuristicas mas nada de muito conclusivo.

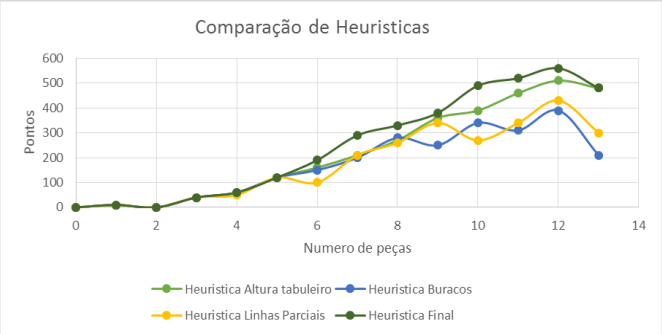

Figura 11: Comparação das heurísticas para um tabuleiro parcialmente preenchido.

Na figura 12, o caso de teste é igual no entanto, o tabuleiro usado estava inicialmente vazio. Podemos observar mais uma vez que ao usar a heuristica que junta todas as outras conseguimos uma melhor pontuação.

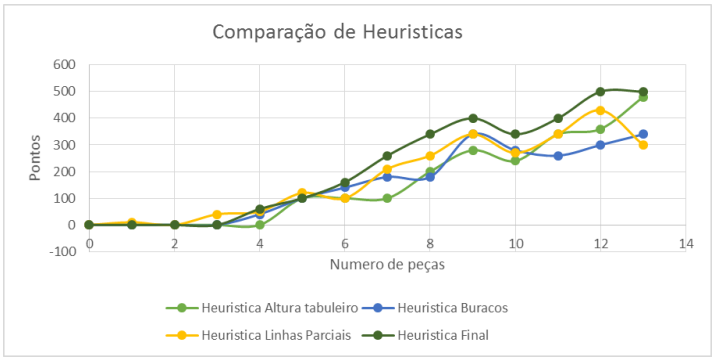

Figura 12: Comparação das heurísticas para um tabuleiro vazio.

4.2.4 Comparação dos Resultados Obtidos

Como se pode verificar na análise dos dados da secção anterior, a procura PpreNivel é aquela que maximiza a pontuação tendo um número de nós gerados expandidos e gerados razoável sendo por isso eficiente em memória e tempo.

As heurísticas por si só conseguem alcançar uma qualidade razoável mas a junção delas fica ainda melhor. A escolha da heurística seria então a heurística final que é uma combinação de várias heurísticas. Todas as comparações já foram efectuadas na secção anterior ao mostrar os resultados obtidos.

4.3 Escolha da procura-best

Para a função procura-best escolhemos a procura PreNivel pois, conforme testado acima, mostra ser a mais vantajosa no contexto do problema.

As heurísticas escolhidas e respetivos coeficiente foram:

- Heurística-altura-tabuleiro-máxima, 11
- Heurística-linhas-parciais, 0.10
- Heurística-buracos, 0.33
- Máximo-pontos-estado, 5.6

As escolhas para estes coeficientes foram tomadas com base no algoritmo genético.

5 Outras ferramentas usadas

Para além das ferramentas usadas, sentimos a necessidade de criar mais algumas adicionais para não só para melhorar a performance do agente, mas também perceber como este funciona, de modo a saber o que seria necessário para o melhorar.

5.1 Algoritmo Genético

O algoritmo gera, para a primeira geração, um conjunto aleatório de genes para cada individuo, em que cada gene corresponde a um coeficiente usado na função heuristica. É então executado uma procura para cada individuo na população tendo em conta os seus genes e a sua perfomance é avaliada segundo uma função de fitness definida pelo utilizador. Esta procura é feita com um estado inicial dado por um gerador de estados. Isto foi necessário após vermos que apenas um estado incial para todas as gerações resultava numa convergência para um genoma especifico para esse problema e portanto em individuos pouco adptaveis.

Os individuos mais aptos são então escolhidos com maior probabilidade para a fase de crossover onde dois pais dão origem a um novo individuo. Este, pela nossa implementação, tem 33% de chance de ficar com o gene do pai A, 33% para o pai B e 33% de ficar com uma média aleatoriamente pesada dos dois pais, sendo isto feito para cada gene indepedentemente. Isto garante que as populações convirgam para uma população apta para o ambiente. Foi também implementada uma fase de mutação que simplesmente modifica um gene para um valor aleatório entre 0 e 1, dando assim alguma diversidade aos individuos dentro de uma geração.

Após a fase de crossover e mutação temos então uma nova população e é executado mais uma nova iteração do algoritmo genético. Isto é feito até antigir um limite de gerações.

Este algoritmo poderá ser analisado na secção Anexo.

5.1.1 Otimizar para uma sequência de peças

Inicialmente, corremos o algoritmo para um jogo de 50 peças gerado aleatoriamente no inicio. Este jogo sofreu 11 gerações com uma população de 50 por geração. Como é possivel verificar no anexo auxiliar no ficheiro 'geneticAlgorithmForFixedPieces', o valor da função fitness (que devolve o quadrado da pontuação obtida), vai aumentando de geração para geração.

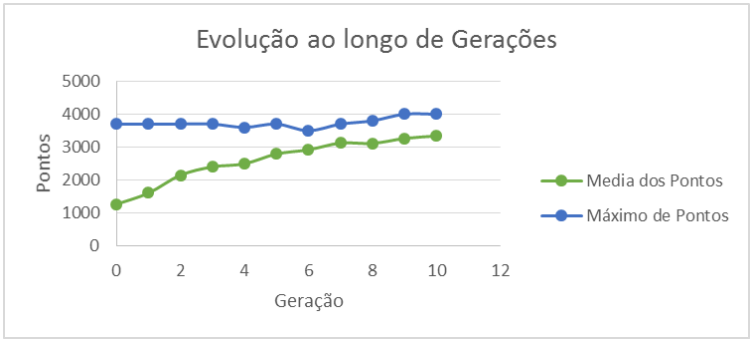

Figura 13: Evolução da qualidade das jogadas ao longo das gerações.

Após análise da figura 13, é possível verificar a evolução do agente ao longo do algoritmo genético. Tendo em conta que no melhor caso possível, faria uma pontuação de 4000 pontos, ao aproximar-se do fim do algoritmo, o agente mostra conseguir uma pontuação cada vez maior, conseguindo até obter a pontuação máxima possível em alguns jogos. Apesar dos bons resultados obtidos, reparámos que, ao correr jogos com peças geradas aleatóriamente, o algoritmo não apresentava os resultados da qualidade esperava. Isto deve-se ao facto de que o agente apenas ficou otimizado para a sequência de peças/tabuleiro gerado no ínicio e portanto tinha um genoma demasiado específico.

5.1.2 Otimizar para uma sequência estocástica de peças

Devido ao problema descrito no final 5.1.1, era necessário melhorar o agente para lidar com sequências de peças aleatórias. Para tal, alterámos o algoritmo genético, para gerar novas sequências de peças a cada geração. Desta forma, o agente está a ser otimizado para peças aleatorias.

A figura 14 mostra a visualização dos dados presentes no anexo auxiliar no ficheiro 'geneticAlgorithmForRandomPieces'. Através dele verifica-se novamente que há uma clara subida na media por geração. Isto deve-se ao facto de o algoritmo genético à medida que corre, faz o agente converger os seus resultados.

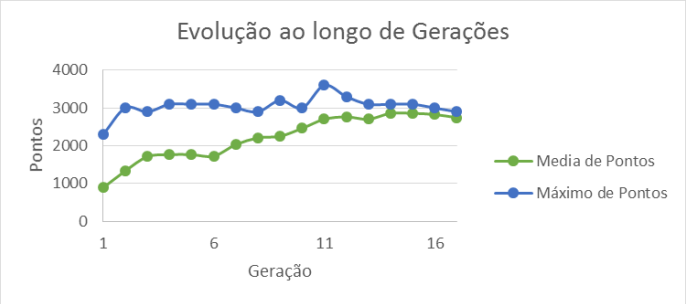

Figura 14: Evolução da qualidade das jogadas ao longo das gerações.

5.1.3 Otimizar para uma sequência estocástica de 6 peças

Após a informação dada sobre a forma de teste da competição, tivemos de alterar a maneira como o agente foi otimizado. Para isso, alterámos novamente o algoritmo genetico de forma a apenas fornecer ao agente 6 peças e um tabuleiro parcialmente preenchido de uma forma aleatória.

Observando a figura 15 e o anexo auxiliar 'geneticAlgorithmFor6RandomPieces', verifica-se mais uma vez como o agente melhora ao longo das gerações e que na geração final, ele consegue quase sempre 300 pontos.

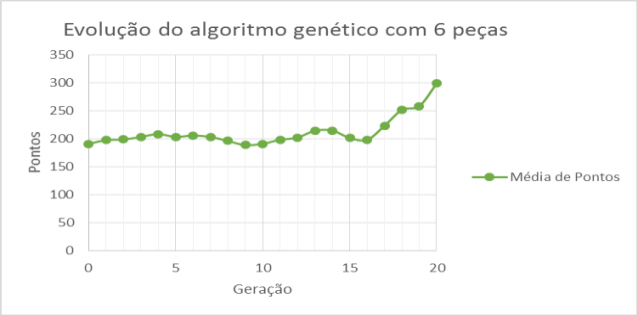

Figura 15: Evolução da qualidade das jogadas ao longo das gerações para 6 peças.

5.2 Visualizar o agente a jogar

Mais importante que o resultado obtido, é saber o que o fez chegar lá de modo a saber como o agente pensa e como o melhorar. Para este objetivo, criámos uma função que permite guardar cada jogada efetuada (Anexo 6.2) para o qual desenvolvemos uma pequena aplicação escrita na linguagem python que nos permite visualizar não só as jogadas do agente mas também valor heuristicos (Anexo 6.2). Isto mostrou-se crucial no decorrer do projeto pois, mais importante que o resultado, é o caminho que percorreu. Um exemplo pode ser observado em http://tinyurl.com/agenteTetris.

6 Anexo

6.1 Algoritmo Genético

```
(defun get_genetic_coefficients (procura gerador-estados solucao resultado fitness heuristicas
                                 &key (constant-h #'(lambda (e) 0)) (pop 100) (genes NIL) (generation 0) (limit 10) (mutation_chance 0.02))
  ;initialize genes as random numbers between -1 and 1
  (when (null genes)
    (setf genes (make-array pop :initial-element NIL))
    (dotimes (i pop)
      (dotimes (j (list-length heuristicas))
        (setf (aref genes i) (cons (- ( / (random 2000) 1000) 1) (aref genes i)))
        )
      )
    )
  (let
    ((lista_accoes NIL) (estado-inicial (funcall gerador-estados)) (current NIL) (fitness_vals (make-array pop))
     (total_fitness 0) (num_h (list-length heuristicas)) (new_genes NIL))
    ;fitness_vals - array of lists: first member -> fitness value, second member -> individual ID (genes index)
    ;new_genes - array containing the genes of the new population
    ;num_h - number of heuristics
    ;total_fitness - total fitness
    (setf new_genes (make-array (list pop)))
    ;get fitness values for all population
    (dotimes (i pop)
      (setf lista_accoes (funcall procura (make-problema :estado-inicial estado-inicial)
                                  #'(lambda (estado) ;Linear function combining all heuristics
                                      (let ((resultado (funcall constant-h estado)))
                                        (dotimes (k num_h resultado)
                                          (setf resultado (+ resultado (* (nth k (aref genes i)) (funcall (nth k heuristicas) estado))))
                                          )
                                        )
                                      ))) ;search for the solution
      (setf current estado-inicial)
      (loop (if (not(and (not (null lista_accoes)) (not (funcall solucao current)))) (return)) ;get final state
        (setf current (funcall resultado current (first lista_accoes)))
        (setf lista_accoes (rest lista_accoes))
        )
      (setf (aref fitness_vals i) (list 0 i)) ;initialize fitness_vals
      (setf (first (aref fitness_vals i)) (funcall fitness current)) ;get fitness value for solution state
      (setf total_fitness (+ total_fitness (first (aref fitness_vals i))))
      (format t "FITNESS: ~$~%" (first (aref fitness_vals i))
      )
    ;sort fitness values
    (setf fitness_vals (sort fitness_vals #'> :key #'(lambda (x)
                                                       (first x)))) ;sort
    (format t "Best: ~$ with genes: ~a" (first (aref fitness_vals 0)) (aref genes (nth 1 (aref fitness_vals 0))))
    (if (= total_fitness 0) (progn
                              (setf total_fitness pop)
                              (dotimes (i pop)
      (setf (first (aref fitness_vals i)) 1)
      )))

    (let ((cumulative_fitness 0)) ;get cumulative probabilities according to fitness value
      (dotimes (i pop)
        (setf cumulative_fitness (+ cumulative_fitness (/ (first (aref fitness_vals i)) total_fitness)))
        (setf (first (aref fitness_vals i)) cumulative_fitness)
        )
      )

    ;select individuals for breeding :D and mutate them D:
    (dotimes (i pop)
      (let ((r1 (/ (random total_fitness) total_fitness)) (r2  (/ (random total_fitness) total_fitness))
            (m (/ (random 1000) 1000))(a 0) (b 0))
        (loop (if (not(and (< a (- pop 1)) (< (first (aref fitness_vals a)) r1))) (return)) ;get first individual
          (incf a))
        (loop (if(not(and (< b (- pop 1)) (< (first (aref fitness_vals b)) r2)))(return)) ;gets second invidivual
          (incf b))

        ;generates a new child from parent A and B
        (setf (aref new_genes i) (crossover (aref genes (nth 1 (aref fitness_vals a)))
                                            (aref genes (nth 1 (aref fitness_vals b)))))
        (if (< m mutation_chance)
            (mutation (aref new_genes i))
          )
        )
      )
    ;returns best solution if limit has been reached or goes to next generation
    (if (>= generation limit) (aref genes (nth 1 (aref fitness_vals 0)))
      (get_genetic_coefficients procura gerador-estados solucao resultado fitness heuristicas
                                :constant-h constant-h :pop pop :genes new_genes :generation (1+ generation) :limit limit)
      )
    )
  )
```

```
(defun crossover (ind_a ind_b)
  (let ((result (make-list (list-length ind_a))))
    (dotimes (i (list-length ind_a) result)
      (if (eq (random 3) 0)
          (setf (nth i result) (nth i ind_a))

        (if (eq (random 2) 0)
            (setf (nth i result) (nth i ind_b))
          (let
              ((r (/ (random 1000) 1000)))
            (setf (nth i result) (+ (* r (nth i ind_a)) (* (- 1 r) (nth i ind_b))))
            )
          )
        )
      )
    )
  )

(defun mutation (genes)
  (setf (nth (random (list-length genes)) genes ) (- (/ (random 2000) 1000) 1))
  )
```

6.2 Algoritmo para visualizar agente a jogar

Função procura-e-guarda, responsável por produzir o output que irá ser utilizador na aplicação de python.

```
(defun procura-e-guarda (procura problema &optional (heuristica NIL))
  (let
      ((lista-accoes NIL) (current (problema-estado-inicial problema))
       (boards "[ [") (heuristics "[") (scores "["))
    (if (null heuristica)     (setf lista-accoes (funcall procura problema ))
      (setf lista-accoes (print (time (funcall procura problema heuristica)))))
    (if (null heuristica) (setf heuristica #'(lambda (e) 0)))
        (dotimes (i 18)
          (setf boards (concatenate 'string boards " [" ))
            (dotimes (j 10)
              (setf boards (concatenate 'string boards (write-to-string (aref (estado-tabuleiro current) i j))))
              (if (< j 9)
                  (setf boards (concatenate 'string boards " ," )))
              )
            (setf boards (concatenate 'string boards "] " ))
            (if (< i 17)
                (setf boards (concatenate 'string boards " ," ))
              (setf boards (concatenate 'string boards " ]" ))))
      (setf scores (concatenate 'string scores "0"))
      (setf heuristics (concatenate 'string heuristics (write-to-string (funcall heuristica current))))
      (loop (if (null lista-accoes)(return))
        (setf current (funcall (problema-resultado problema) current (first lista-accoes)))
        (setf boards (concatenate 'string boards ", ["))
        (dotimes (i 18)
          (setf boards (concatenate 'string boards " [" ))
            (dotimes (j 10)
              (setf boards (concatenate 'string boards (write-to-string (aref (estado-tabuleiro current) i j))))
              (if (< j 9)
                  (setf boards (concatenate 'string boards " ," )))
              )
            (setf boards (concatenate 'string boards "] " ))
            (if (< i 17)
                (setf boards (concatenate 'string boards " ," ))
              (setf boards (concatenate 'string boards " ]" ))))
        (setf scores (concatenate 'string scores ", " (write-to-string (estado-pontos current))))
        (setf heuristics (concatenate 'string heuristics ", " (write-to-string (funcall heuristica current))))
        (pop lista-accoes)
        )
      (setf boards (concatenate 'string boards "]"))
      (setf scores (concatenate 'string scores "]"))
      (setf heuristics (concatenate 'string heuristics "]"))
      (format t "~$" (concatenate 'string boards ", " scores ", " heuristics ", "))
      )
    )
```

Aplicação desenvolvida em *python* que permite uma representação visual das acções do agente.

```
import simplegui
import random
W = 10
H = 18
window_h = H*10+60
window_w = W*10 + 50
color_new = "White"
color_old = "Blue"
color_unfilled = "Gray"

scores = [0 for k in xrange(10)]
heuristics = [0 for k in xrange(10)]
old_board = [ [False for i in xrange(H)] for j in xrange(W)]
boards = [[
        [(False, True)[random.randint(0,1)]
         for i in xrange(H)]
        for j in xrange(W)]
          for k in xrange(10)]
NIL = False
T = True
boards, scores, heuristics = #output de procura-e-guarda
final_cost = scores[-1]
actual_heuristic = [final_cost - score for score in scores]

i=0
def timer_handler():
    global i, old_board
    old_board = boards[i]
    if i<len(boards)-1:
        i = i+1
# Handler for mouse click
def click():
    if timer.is_running():
        timer.stop()
    else:
        timer.start()

def previous():
    global i, old_board
    if i>0:
        i=i-1
    old_board = boards[max(0, i-1)]
def next2():
    global i, old_board
    if i<len(boards)-1:
        i=i+1
    old_board = boards[i-1]

def draw_board(board, canvas):
    global old_board
    for i in xrange(W):
        for j in xrange(H):
            a=i*10
            b=window_h-j*10 - 10
            if board[j][i]:
                if old_board[j][i]:
                    color = color_old
                else:
                    color = color_new
            else:
                    color = color_unfilled
            canvas.draw_polygon([(a, b),(a+10, b), (a+10, b+10),(a, b+10)], 1,"Black",color)

# Handler to draw on canvas
def draw(canvas):
    draw_board(boards[i], canvas)
    canvas.draw_text('Score: ' + str(scores[i]), [1, 10], 10, 'White')
    canvas.draw_text('Heuristic: ' + str(heuristics[i]), [1, 20], 10, 'White')
    canvas.draw_text('Actual Heuristic: ' + str(actual_heuristic[i]), [1, 30], 10, 'White')
    #canvas.draw_polygon([(10, 20), (20, 30), (30, 10)], 12, 'Green')

# Create a frame and assign callbacks to event handlers
frame = simplegui.create_frame("Home", window_w, window_h)
frame.add_button("Pause/Resume", click)
frame.add_button("Previous", previous)
frame.add_button("Next", next2)
frame.set_draw_handler(draw)

timer = simplegui.create_timer(1000, timer_handler)
```

6.3 Anexo Auxiliar

Devido ao tipo de conteúdo, em vez de por no relatório os resultados do algoritmo genético diretamente no relatório, estes encontram-se numa pasta Dropbox de acesso público.

Basta aceder ao seguinte link: http://tinyurl.com/projetoIA