MAC0425 – Inteligência Artificial

Mina de Ouro

Daniel Augusto Cortez

28 de setembro de 2013

Resumo

Relatório descrevendo implementação e resultados do EP1 de Inteligência Artificial, IME-USP 2013 (Mina de Ouro).

1 Introdução

Este relatório descreve a minha implementação do EP para resolver o problema da mina de ouro proposto no enunciado, bem como oferece informações sobre testes e análise de desempenho das diferentes buscas.

A implementação foi escrita em Java (1.6.29) no ambiente de desenvolvimento Eclipse (Kepler) utilizando o sistema operacional Mac OS (10.7.2).

O código fonte (classes Java) se encontra no diretório `/src/dacortez/minaDeOuro`. Uma versão compilada do programa `minaDeOuro.jar` está disponível no diretório raiz junto com alguns arquivos de entrada. A utilização do programa se faz através da linha de comando:

```
$ java -jar minaDeOuro.jar <arquivo_de_entrada> <tipo_de_busca>
```

Os tipos de busca suportados são:

```
P: busca em profundidade limitada
L: busca em largura
A: busca A*
U: busca uniforme
```

2 Estrutura de Classes

A implementação foi realizada modularizando o programa em 15 classes, conforme descrição abaixo (com comentários sobre os principais métodos de cada classe). A documentação completa (Javadoc) está disponível no diretório `/doc`.

- `Main.java`: Ponto de entrada do programa. Cria o objeto `Enviroment` apropriado a partir do arquivo de entrada e instância o tipo de agente escolhido. O objeto `Environment` pode ser acessado estaticamente pois é único ao longo da vida do agente. Os tipos de agente que podem ser instanciados efetuam busca em largura limitada, busca em profundidade, busca A* e busca uniforme.

- `Environment.java`: Representa o ambiente da mina, contendo um mapa com suas posições livres, obstruídas e com pepitas de ouro. Possui métodos que permitem ao agente decidir como se mover ou se é possível pegar ouro. O método `performanceMeasurement()` avalia a performance do agente ao tomar a ação passada. Caso a ação seja pegar ouro, retorna $4n$, onde n é a dimensão da mina. Caso contrário, retorna -1.

- `Agent.java`: O agente é responsável por efetuar o procedimento de busca adequado na mina. O método de busca da solução `getSolution()` é abstrato e deve ser implementado pela instância concreta do agente que deriva desta classe. A estrutura comum a todos esses agentes, entretanto, estão presentes nesta classe base.

  O principal método do agente é `search()` que, a partir da estratégia de busca do agente, explora a mina tentando coletar $1, 2, \ldots,$ todas as pepitas da mina, retornando a melhor solução possível.

- `AStarAgent.java`: Este agente implementa o método de busca A*. É uma classe abstrata dericada de `Agent`. As classes concretas devem implementar a função heurística. A implementação do método `getSolution()` é baseada no código `GRAPH-SEARCH` de [1] utilizando uma fila de prioridades para os nós baseada na alaviação da função $f(n) = g(n) + h(n)$.

- `AStarAgentHNearst.java`: Concretiza a classe `AStarAgent` utilizando uma heurística que encontra as pepitas mais próximas do agente, uma a uma, e depois retorna à posição inicial. Falaremos mais sobre a heurística utilizada na Seção 3.

- `AStarAgentHZero.java`: Concretiza a classe `AStarAgent` utilizando uma heurística nula, o que equivale fazer a busca uniforme. Ou seja, esse agente implementa a busca uniforme.

- `BreadthAgent.java`: Concretiza a classe `Agent` implementando o método de busca em largura. A implementação do método `getSolution()` é baseada no código `GRAPH-SEARCH` de [1] utilizando uma fila FIFO padrão.

- `LimitedDepthAgent.java`: Este agente implementa o método de busca em profundidade limitada. A implementação do método `getSolution()` é baseada no código recursivo `DEPTH-LIMITED-SEARCH` de [1].

- `Position.java`: Representa uma posição na mina como o par ordenado (linha, coluna). A posição (0, 0) corresponde ao canto superior esquerdo. Os valores de linha crescem para baixo e os valores de coluna crescem para direita. Sobrescreve o método `equals()` para se poder comparar instâncias diferentes de posições pelos valores de linha e coluna.

- `Action.java`: Enum que define as ações que o agente pode executar na mina:

  ```
  RIGHT: mover para direita.
   LEFT: mover para esquerda.
   DOWN: mover para baixo.
   PICK: pegar ouro.
  ```

  O método `toString()` retorna uma String descritiva da ação para ser utlizada na impressão da solução.

- `State.java`: Representa o estado onde se encontra o agente em sua busca. O estado é representado pela posição do agente e uma lista das posições das pepitas de ouro recolhidas. O principal método da classe é `getSuccessors()` que representa a função sucessora para o estado atual. O método `equals()` foi sobrescrito para permitir comparação entre instâncias diferentes de estado baseada na posição do agente e na lista de pepitas coletadas.

- `ActionState.java`: Representa um par (action, state), onde o estado state é atingido após a execução da ação action.

- `Node.java`: Representa um nó da árvore de busca expandida pelo agente. Os atríbutos são o estado, a ação que levou a este nó, o nó pai, o custo do caminho até o nó e a sua profundidade. O principal método da classe é `expand()`, que retorna uma lista com todos os nós que podem ser obtidos a partir das possíveis ações do agente sobre o estado do nó atual. (código baseado na função `EXPAND` de [1])

- `Solution.java`: Representa a solução encontrada pelo agente de busca, contendo o caminho total entre o nó raiz e o nó objetivo. O método `toString()` foi sobrescrito para retornar a solução com a pontuação do agente e o plano de ações.

- `Cutoff.java`: Esta classe extende `Solution` apenas para ser utilizada no método de busca em profundidade limitada para indicar que o limite da busca foi atingido, porém a solução procurada ainda não foi encontrada.

A hierarquia de classes entre os agentes é um pouco mais complicada, mas bastante natural. Um diagrama UML ilustrativo é apresentado no Figura 2.

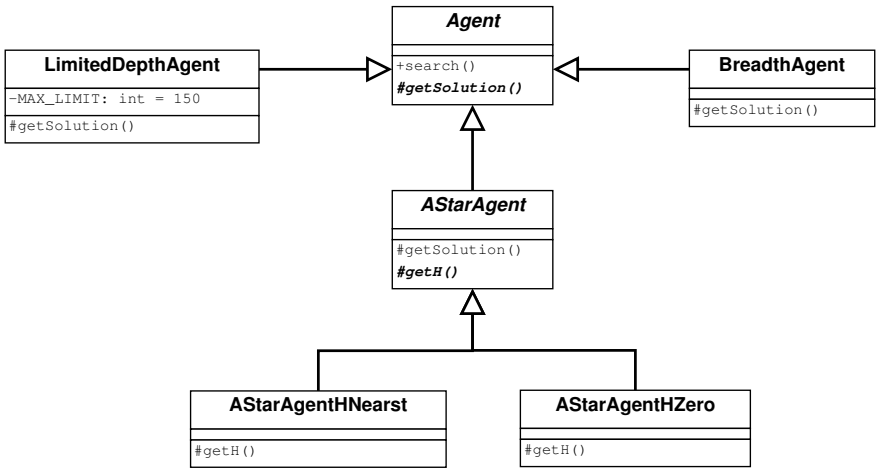

Figura 2.1: Digrama UML simplificado representando a hierarquia de classes entre os agentes implementados.

3 Heurística Utilizada

A heurística utilizada na implementação do método de busca do agente A* consiste na ideia natural de se calcular o número de passos que seriam necessários para o agente coletar a pepita mais próxima a ele, depois seguir dessa posição para a próxima pepita que esteja mais perto e assim sucessivamente até que se colete todas as pepitas desejadas, retornando finalmente à posição inicial da mina (confira a Figura 3). Esse número total de passos entraria com sinal negativo e o total de ouro coletado (vezes $4n$) entraria com sinal positivo para dar o valor da função heurística:

$$h(n) = -\sum(\text{passos até pepitas mais próxima}) + 4n\sum(\text{pepitas})\,.$$

O número de passos entre duas posições é calculado considerando que não existe nenhuma obstrução na mina entre elas, sendo portanto a soma das diferenças entre linhas e colunas. Se x e y são duas posições na mina, então definimos a distância entre elas como:

$$\|x - y\| = |\operatorname{row}(x) - \operatorname{row}(y)| + |\operatorname{col}(x) - \operatorname{col}(y)|\,.$$

Pelo fato de estarmos calculando distâncias diretas entre pontos na mina sem considerar as obstruções e estarmos usando uma estratégia gulosa (ir atrás da próxima pepita que esteja mais perto), a heurística definida deve ser adimissível[1].

4 Testes e Análise de Desempenho

O programa foi testado usando 6 arquivos de entrada (os arquivos estão no diretório raiz numerados de um a seis) representando minas com dimensões diferentes. Medimos a pontuação obtida por cada um dos quatro agentes implementados, bem como o tempo de processamento. Os resultados são resumidos na Tabela 4.1.

Os resultados da tabela claramente indicam que a busca em profundidade não obtém resultados ótimos mesmo para instâncias pequenas. Apesar disso seu tempo de processamento é baixo. A busca em largura obtém resultados ótimos mais é a que consome mais tempo para instâncias maiores. A busca uniforme e a busca A* são mais rápidas para instâncias maiores mas nem

[1]Confira a Seção 5 para uma discussão sobre isso

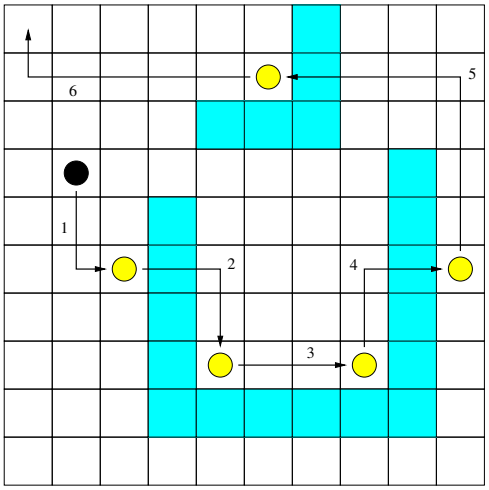

Figura 3.2: Representação da heurística utilizada. O agente (ponto preto) seguiria os caminhos 1, 2, 3, 4, 5 e 6 coletando todas as pepitas (pontos amarelos) e retornando a sua posição inicial. Note que não há preocupação com as obstruções dos caminhos (quadrados em azul).

n	**P**	**L**	**U**	**A**
6	20 (0.006)	34 (0.021)	34 (0.020)	34 (0.065)
8	56 (0.029)	70 (0.092)	70 (0.067)	70 (0.131)
10	66 (0.095)	118 (0.154)	118 (0.109)	118 (0.209)
14	194 (2.288)	306 (8.842)	298 (0.355)	304 (0.638)
16	192 (16.497)	412 (84.34)	406 (3.895)	412 (1.120)
20	0 (15.963)	–	–	726 (1.431)

Tabela 4.1: Pontuação obtida e tempo de processamento (entre parenteses em segundos) para cada um dos agentes implementados e para as seis entradas consideradas, representadas pela dimensão da mina n. P = busca em profundidade, L = busca em largura, U = busca uniforme, A = busca A*. Os traços indicam que nenhuma solução foi obtida com até 180 segundos de processamento.

sempre obtém o ótimo. No caso $n = 16$, a busca A* atingiu o ótimo com o tempo de processamento bem inferior do que o da busca em largura. No caso $n = 20$ apenas a busca A* conseguiu um resultado satisfatório.

5 Conclusões

A implementação apresentada é baseada em bom projeto de classes, conforme apresentado na Seção 2. Os resultados da Table 4.1 indicam que os agentes se comportam de maneira esperada, sendo A* o mais eficiente para instâncias maiores.

Alguns pontos devem ser considerados. Primeiro note que a busca do agente não é realizada com passos de mesmo custo, pois em alguns casos ele perde um ponto ao se mover e, em outro, ganha $4n$ pontos ao coletar uma pepita. Nesse caso [1], nem mesmo a busca em largura tem garantia de ser ótima. A busca uniforme, entretanto, deveria ser. Mas como a implementação evita estados repetidos, utilizando o algoritmo `GRAPH-SEARCH` de [1], nesse caso, também não se tem a garantia de otimalidade. O mesmo acontece para a busca em profundidade, que é bem ineficiente dado que limitou-se a profundidade máxima para um valor baixo (`MAX_LIMIT = 150`)a fim de se evitar tempos excessivos de processamento.

A busca A*, entretanto, deve ser ótima para `GRAPH-SEARCH` desde que a heurística $h(n)$ seja consistente. O fato de termos encontrado um resultado não ótimo para a instância $n = 14$ (304 pontos contra 306 da busca em largura) indica que a heurística construída não é consistente e, portanto, também não deve ser adimissível.

Referências

[1] S. Russell, P. Norvig. *Artificial Intelligence – A Modern Approach*. Segunda edição.