Atividade Prática 1: Quebra-cabeça de oito peças

Inteligência Artificial

Raoni F. S. Teixeira

Introdução

Este documento é parte da primeira avaliação prática da disciplina de inteligência artificial ministrada na UFMT no segundo semestre de 2016. Nesta atividade, você irá implementar o algoritmo de busca heurística A* (a.k.a., A-estrela) para resolver o quebra-cabeça de 8 peças. Lembre-se de que esta é uma atividade individual e é muito importante que você escreva a sua própria solução e o seu próprio relatório.

Antes de iniciar o exercício, recomendamos fortemente a leitura atenta dos Capítulos 2 e 3 do livro texto [1] e a consulta ao material sobre Octave/MATLAB disponível na página da disciplina [2].

Arquivos incluídos na atividade

- `ex1.m` Script Octave/Matlab que orientará você no exercício.
- `aquecimento.m` Script que guiará você na tarefa de aquecimento.
- `extra1.m` Script que orientará você nas tarefas opcionais.
- `PriorityQueue.m` Implementação da fila de prioridades.

[1] Stuart Russell e Peter Norvig. Inteligência Artificial. Tradução da 3a edição. Editora Campus/Elsevier.

[2] `http://www.students.ic.unicamp.br/~ra089067/ensino/2016_2/ia.html`

- `legal_moves.m` Função que devolve os movimentos válidos em um determinado estado do jogo.
- `do_move.m` Função que movimenta uma peça e devolve a matriz resultante.
- `hamming.m` Função que calcula a quantidade de células na posição errada para um determinado estado do jogo.
- `reconstruct_path.m` Função que mostra os movimentos realizados.
- `contagem.m`$^+$ Função simples que conta a quantidade de células de uma matriz tem valor igual a um.
- `manhattan.m`$^+$ Função que implementa a heurística de manhattan.
- `astar.m`$^+$ Função que implementa o algoritmo A*.
- `heuristic.m`$^+$ Função que define uma nova heurística para o problema.

Todos os arquivos marcados com $^+$ devem ser implementados (alterados).

---

1 Aquecimento

A primeira parte deste exercício trata da sintaxe do Octave/MATLAB. A ideia é que, após esta pequena tarefa, você terá mais intimidade com a linguagem e com ambiente.

No arquivo `contagem.m`, você encontrará o rascunho de uma função em Octave/MATLAB. Esta função deve receber como parâmetro uma matriz M e calcular a quantidade de elementos de M com valor igual a 1 (um). Quando você terminar, execute o script `aquecimento.m` e teste a função criada com diferentes matrizes.

2 Agentes baseados em busca

Na segunda parte da atividade, você irá implementar um agente computacional racional. Nesta seção, discutimos alguns aspectos envolvidos neste tipo

de tarefa. Esta questão é central em toda inteligência artificial. Utilize estas definições como suporte para elaboração da solução desta e das próximas atividades.

2.1 Definições preliminares

Um *agente* é tudo o que pode ser considerado capaz de **perceber** seu ambiente por meio de sensores e de **agir** sobre esse ambiente por intermédio de atuadores. Mais formalmente, pode-se dizer que um agente é uma função f que mapeia cada possível sequência de percepções para uma ação, da seguinte forma:

$$f : \mathcal{P}^* \mapsto \mathcal{A}, \tag{1}$$

em que $\mathcal{P}$ indica o conjunto de percepções e $\mathcal{A}$ representa o conjunto de ações.

É possível avaliar um agente de diferentes formas. Pode-se considerar, por exemplo, uma perspectiva operacional que leva em conta as *consequências* de seu comportamento. Neste caso, dizemos que um agente tem um *bom comportamento*, se a sequência de ações escolhidas, faz com que ele passe por uma sequência *desejável* de estados do ambiente. A cada nova ação executada pelo agente, o ambiente (e.g., localização do agente, características tais como limpeza ou iluminação) pode mudar. Este tipo de agente (a.k.a, **agente racional**) será amplamente estudado neste curso.

A noção de *desejável* enunciada acima pode ser capturada por uma **medida de desempenho** que avalia qualquer sequência de estados do ambiente. Em muitos casos, esta medida de desempenho pode ser definida como uma função g que mapeia cada estado do ambiente (ou, mais especificamente, o seu produto cartesiano com as possíveis sequências de ações) para um número real, da seguinte maneira:

$$g : \mathcal{S}^* \times \mathcal{A}^* \mapsto [-\infty, \infty], \tag{2}$$

em que $\mathcal{S}$ corresponde ao conjunto de estados do ambiente. Assim, é possível avaliar (dar uma nota) cada ação executada pelo agente. Um **agente racional** é uma função f que otimiza g. De uma maneira um pouco mais informal (e seguindo o livro texto), podemos dizer isto de outra maneira, mais ou menos assim:

> ... um agente racional seleciona, para cada sequência de percepções possível, uma ação que maximiza a medida de desempenho esperado, dada a evidência fornecida pela sequência de percepções e por qualquer conhecimento interno seu.

Para terminar, é importante dizer que esta definição não produz necessariamente um agente perfeito. A racionalidade otimiza o desempenho esperado. A perfeição, por outro lado, otimizaria o desempenho real (infelizmente, um agente perfeito precisaria ser também *onisciente* - consulte o livro texto para mais detalhes).

2.2 Resolução de problemas com busca

Em algumas casos, é possível simplificar ainda um pouco mais a definição anterior. Vamos considerar primeiro uma simplificação que pode ser adotada sempre que o ambiente for:

- *observável*: o agente sempre conhece o seu estado atual;
- *discreto*: em qualquer estado, há sempre um número finito de ações a serem executadas;
- *conhecido*: o agente sabe exatamente qual estado será alcançado quando uma ação for executada e
- *determinístico*: cada ação produz exatamente um resultado.

Também vamos adotar um critério de *racionalidade* focado especificamente nas ações que um agente deve executar para alcançar um determinado objetivo (i.e., estados do ambiente com um solução do problema). Sob tais circunstâncias, uma *solução* (ou uma das soluções) para o problema é uma sequência fixa de ações. Note que como o agente sabe com antecedência qual estado será alcançado para cada ação, ele pode ignorar as informações percebidas através do sensor [3].

Com estas restrições em mente, podemos reformular a definição anterior de agente racional. Esta nova definição baseia-se em 5 (cinco) componentes importantes. O primeiro deles é o estado inicial $s_i \in \mathcal{S}$ em que o agente

[3] Os teóricos da teoria de controle chamam isto de sistema *malha aberta*.

começa. Outro componente importante é a descrição das ações possíveis para cada estado. Formalmente, esta descrição pode ser representada por uma função A, $A(s) \mapsto \mathcal{A}' \subset \mathcal{A}$, definida para qualquer estado $s \in \mathcal{S}$. Também precisamos de um modelo de transição T, $T(s, a) \mapsto s'$, especificando o estado que será alcançado (s') quando uma determinada ação a for executada em qualquer estado $s \in \mathcal{S}$. Precisamos ainda de conjunto de estados objetivos $\mathcal{O}$, $\mathcal{O} \subset \mathcal{S}$, e de uma função c, $c(a_1, a_2, \ldots, a_n) \mapsto [-\infty, \infty]$, que calcula o custo de toda trajetória de ações que pode ser executada pelo agente. Quanto maior o custo, pior é a trajetória.

Podemos então concluir que:

> ... [neste contexto] um agente pode ser chamado de *racional* sempre que for capaz de encontrar uma sequência de ações que leva ao estado objetivo (ou um dos estados objetivo) usando o trajeto (ou um dos trajetos) com menor custo.

Com um pouco de atenção, é possível perceber que um algoritmo de busca pode ser utilizado para resolver esta nova formulação do problema. Podemos, por exemplo, construir uma árvore de busca cujos nós correspondem aos estados do ambiente e a raiz representa o estado inicial s_i. Os ramos da árvore (interligação entre os nós) indicam as ações que podem ser executadas a partir de um determinado estado. A essência desta estratégia é simples: tentamos seguir uma determinada trajetória (i.e., sequência de ações) e deixamos outras trajetórias reservadas para mais tarde (estas trajetórias serão testadas apenas se alguma das anteriores não for uma solução razoável para o problema). Se o problema tiver solução, nós a encontraremos.

Este tipo de problema foi exaustivamente estudado pela comunidade de computação. Há uma série de algoritmos que podem ser considerados para resolvê-lo (e.g., busca em largura, busca em profundidade). Como vimos na aula, tais algoritmos podem ser classificados em 2 (duas) categorias principais:

- busca cega: usa apenas conhecimento sobre a especificação do problema do problema e
- busca heurística (ou busca informada): usa conhecimento heurístico sobre domínio.

Além destas duas categorias, existe ainda um outra categoria chamada de *busca local*. Os algoritmos de busca local, porém, não armazenam in-

formações sobre as trajetórias percorridas e, por este motivo, não podem ser utilizados para encontrar a *sequência de ações* que o agente deve executar. Nesta atividade, você deve implementar um algoritmo de busca heurística (apresentamos uma breve discussão na próxima seção). Detalhes sobre outros algoritmos podem ser encontrados no livro texto.

3 Implementação principal

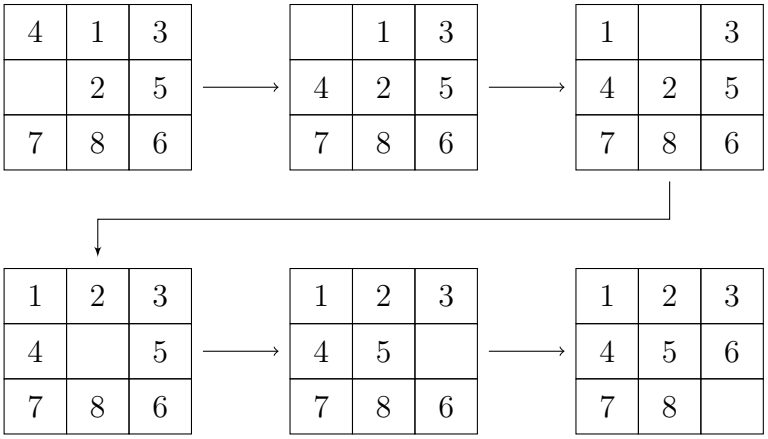
Algoritmos de busca com heurística exploraram o fato de que algum conhecimento extra sobre o domínio do problema pode ser utilizado para guiar a busca em direção à regiões em que (pelo que se sabe) é mais provável encontrar uma solução. Do ponto de vista prático, estes algoritmos usam uma função h que tenta prever o custo de caminhos ainda não visitados. O termo 'heurística' tem portanto o mesmo sabor da expressão *aproximação imperfeita* e é usado para indicar que o algoritmo fará aproximações progressivas até encontrar a solução do problema.

Nesta parte da atividade, você irá implementar o algoritmo A* para resolver o quebra-cabeça de oito peças (consulte o livro texto para mais detalhes). Para tanto, você deve preencher o arquivo `astar.m`. Descrições de aspectos importantes que devem ser considerados nesta implementação são apresentada a seguir.

3.1 Detalhes da problema

O quebra-cabeça de oito peças consiste em uma matriz 3×3 com 8 (oito) células quadradas rotuladas de 1 a 8 e uma célula em branco [4]. O objetivo do jogo é reorganizar as células para que elas fiquem em ordem. É permitido mover as células horizontalmente ou verticalmente (mas não diagonalmente) para o quadrado em branco. Um exemplo de seqüência de movimentos válidos é apresentada a seguir.

[4] No código desta atividade, o branco será representado pelo valor 9.



Analisando atentamente este exemplo, podemos perceber que neste problema um estado é uma configuração específica da matriz. Também podemos identificar os 5 (cinco) componentes fundamentais para construção da solução baseada em busca. Pode-se notar primeiro que qualquer configuração específica da matriz pode ser um estado inicial e que as ações executadas devem ter uma correspondência com os movimentos esquerda, direta, acima ou abaixo que podem ser executados ao redor da célula vazia da matriz. Considerando isto, é fácil definir a descrição de ações e o modelo de transição (consulte os arquivos `legal_moves.m` e `do_move.m`). Para finalizar, podemos perceber que há apenas um estado objetivo (configuração com as células da matriz em ordem) e que a função de custo deve levar em conta os passos percorridos para se chegar ao objetivo. O custo de cada passo é igual a 1 (um) movimento.

No pacote distribuído, a função `legal_moves.m` implementa a descrição de ações e a função `do_move.m` implementa o modelo de transição. Utilizando estas funções, é possível gerar cada estado `N` que é vizinho de um estado `State` da seguinte maneira:

```
State = [4 1 3; 9 2 5; 7 8 6];
moves = legal_moves(State);
for i=1:size(moves, 1),
   N = do_move(State, moves(i));
end
```

3.2 Detalhes da implementação

A solução computacional clássica para este problema consiste em manter em memória os nós visitados pelo algoritmo. Para que seja possível *reconstruir* o caminho percorrido até a solução, cada nó produzido pelo algoritmo de busca deve conter uma referência para o nó que o descobriu (i.e., nó visitado no passo anterior). Podemos descobrir quais ações foram executadas apenas seguindo as referências (consulte o arquivo `reconstruct_path.m`). Também é importante que cada nó armazene o estado atual (i.e., configuração da matriz) e o número de movimentos feitos até o momento. Esta estrutura já está implementada no pacote distribuído com o exercício (consulte o arquivo `Node.m`).

O algoritmo A* utiliza uma fila de prioridades e uma função h, $h(s) \mapsto [-\infty, \infty]$, que tenta prever a quantidade de ações que devem ser realizadas para alcançar o objetivo a partir do estado s em questão. A estratégia é explorar primeiro os nós que representam estados com menor custo esperado, pois eles possivelmente estariam mais próximos do objetivo.

O custo esperado, c, para um nó de busca n que representa um estado s qualquer é a soma entre a quantidade de passos já realizados para chegar em n e a quantidade de passos que ainda faltam para chegar no objetivo. Como não sabemos exatamente quantos passos faltam, utilizamos o resultado da função h como uma estimativa. Juntando tudo, temos que o custo do nó n é dado por:

$$c(n) = f(n) + h(n.s), \tag{3}$$

em que n é um nó de busca representando o estado s.

A fila de prioridades é utilizada para reservar os nós (e caminhos) ainda não visitados pelo algoritmo e para facilitar a escolha do nó mais próximo do objetivo. Obviamente, neste caso a prioridade de cada nó n é definida pela função $c(n)$. No código disponibilizado, há uma implementação de fila de prioridades no arquivo `PriorityQueue.m`.

O algoritmo pode ser descrito da seguinte maneira. Primeiro, crie um nó para o estado inicial e insira este nó em uma fila de prioridade (inicialmente vazia). Em seguida, enquanto a fila não estiver vazia ou um nó com o estado objetivo não for alcançado, remova da fila de prioridade o nó m com a menor prioridade, crie um nó para todos os estados vizinhos de $m.s$ (aqueles que podem ser alcançados com um único movimento a partir de $m.s$) e insira estes nós na fila de prioridade. O procedimento deve ser repetido até que a condição de parada seja satisfeita (i.e., até que fila fique vazia ou o nó com o

estado objetivo seja alcançado).

Este algoritmo deve ser implementado no arquivo `astar.m`. Alguns detalhes que serão úteis para esta implementação são apresentados a seguir. Para criar um novo nó de busca `n` utilizando o pacote distribuído com a atividade, você deve utilizar a instrução `Node` mais ou menos assim:

```
n = Node(State, cost);
```

em que `State` e `cost` correspondem respectivamente ao estado e ao custo do nó. O custo do nó é igual ao número de passos efetivamente percorridos pelo algoritmo até a descoberta do nó `n` (i.e., o custo de um nó não-raiz é 1 (um) mais o custo do nó que o descobriu).

Para cada nó criado, é preciso definir também a referência para o nó que o descobriu. A instrução a seguir indica que o nó `n` foi descoberto pelo nó `p`.

```
n.Prev = p;
```

Note que para acessar os atributos (i.e., estado, custo ou nó anterior) de um nó `n`, você pode utilizar o operador `.` (ponto). Para alterar o estado de um nó `n`, por exemplo, podemos fazer:

```
S = [4 1 3; 9 2 5; 7 8 6];
n.State = S;
```

A fila de prioridades já está implementada no arquivo `PriorityQueue.m` O código a seguir criar uma fila de prioridades `q` vazia.

```
q = PriorityQueue() ;
```

e a instrução

```
q.insert(p, n);
```

insere um nó `n` com prioridade `p` na fila `q`. A prioridade do nó `n` deve ser calculada somando o custo efetivo do nó `n.f` com valor estimado pela função heurística para estado correspondente, assim:

```
p  =  n.f+h(n.State)
```

em que `h` indica uma função heurística (e.g., hamming ou manhattan).

Para remover um nó `m` com a menor prioridade, é só utilizar a função `extracMin` assim:

```
m = q.extractMin();
```

O sucesso do algoritmo de busca depende da função utilizada para prever o caminho que falta para alcançar o objetivo. Nesta atividade, você deve considerar duas abordagens:

- função *hamming*: números de células na posição errada. Esta função já está implementada no pacote distribuído (veja o arquivo `hamming.m`).
- função de *manhattan*: soma das distâncias (distâncias horizontal e vertical) de cada célula para a posição que ela deveria ocupar no estado objetivo. Nesta atividade, você deve criar a sua implementação para esta função e preecher o arquivo `manhattan.m`.

A seguir, ilustramos como estas duas funções funcionam. Observe que a função manhattan considera a soma do deslocamento horizontal e vertical necessário para mover a célula para a posição adequada.

4	1	3
8	2	5
7		6

Estado s

1	2	3
4	5	6
7	8	

Objetivo

Posições

1	2	3	4	5	6	7	8
1	1	0	1	1	1	0	1

Hamming(s) = 6

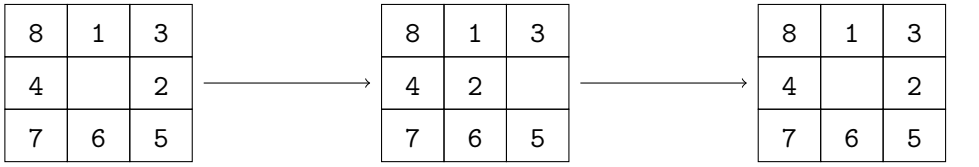
Posições

1	2	3	4	5	6	7	8
1	1	0	1	1	1	0	2

Manhattan(s) = 7

Estas duas funções tem uma característica importante: a quantidade de movimentos necessários para alcançar o objetivo é no mínimo igual ao valor calculado por elas. No caso da função hamming, é fácil perceber que toda célula fora do lugar deve ser movimentada pelo menos uma vez para a posição adequada. No caso da função manhattan, isto também é verdade porque, para alcançar o objetivo, toda célula fora do lugar deve ao menos percorrer a distância horizontal e vertical que a separa do objetivo.

Depois de implementar o algoritmo, você irá perceber um fato irritante: nós correspondendo ao mesmo estado (i.e., com matrizes iguais) são adicionados na fila de prioridades muitas vezes. Para evitar isto, altere o código do algoritmo para que nós com estado visitado no passo anterior não sejam inseridos na fila de prioridades (veja a ilustração a seguir).



Anterior Atual Vizinho (anterior)

Depois de corrigir este problema, teste sua implementação com alguns exemplos simples e, só então, execute o script `ex1.m`.

3.3 Exercício opcional (extra)

Você pode optar por resolver duas questões extras. A primeira delas envolve a identificação de jogos que não possuem solução. Como você talvez tenha observado, nem toda matriz do jogo pode ser reorganizada para que suas células fiquem em ordem. Este é o caso, por exemplo, da matriz ilustada a seguir.

1	2	3
4	5	6
8	7	

Para detectar quando um jogo não tem solução, você pode explorar o fato de que as matrizes deste jogo são divididas em duas classes: i) aquelas que podem ser resolvidas e ii) aquelas que só podem ser resolvidas se qualquer par de células adjacentes e não vazias for trocado. A matriz a seguir (resultado da troca das células de posição das células 5 e 7 no exemplo anterior) agora tem solução. Esta verificação deve ser implementada no arquivo `astar.m`. A função `astar.m` deve devolver um valor (variável `error`) indicando se o jogo tem ou não solução.

1	2	3
4	7	6
8	5	

Na outra tarefa extra da atividade, você é desafiado a criar a sua própria heurística para o problema. A função criada por você deve ter a mesma propriedade das duas funções definidas na seção anterior (i.e., para qualquer estado s a quantidade de movimentos necessários para alcançar o objetivo a partir de s deve ser no mínimo igual ao valor calculado por sua função). Você deve comparar a função criada com as duas heurísticas anteriores e discutir os resultados no relatório. A nova heurística deve ser implementada no arquivo `heuristic.m`.

Para testar sua implementação, execute o arquivo `extra1.m`.

4 Relatório, entrega e notas

Depois de terminar a implementação, você deve escrever um pequeno relatório contendo obrigatoriamente:

- uma seção "Resumo" que deve claramente contextualizar e apresentar os principais resultados do trabalho e
- uma seção "Resultados e discussões" em que os resultados (i.e., tabelas, gráficos) devem ser apresentados e interpretados.

A seção "Resultados e discussões" deve apresentar uma comparação entre o desempenho das funções heurísticas utilizadas.

O relatório e os códigos devem ser entregues até o dia **02 de fevereiro de 2017** (no fim do recesso). A pontuação de cada tarefa deste exercício é apresentada na Tabela 1.

Tarefa	**Arquivo**	**Pontuação (nota)**
Exercício de Aquecimento	`contagem.m`	0.5
Função heurística *Manhattan*	`manhattan.m`	1.0
Algoritmo A*	`astar.m`	5.5
Relatório	`ap1.pdf`	3.0
Detecção de jogos sem resposta	`astar.m`	0.75 (extra)
Definição de outra heurística	`heuristic.m`	1.25 (extra)

Tabela 1: Pontuação de cada tarefa do exercício.