Universidade Federal de Minas Gerais **Inteligência Artificial**
Instituto de Ciências Exatas **2º Semestre de 2015**
Departamento de Ciência da Computação **Prof. Luiz Chaimowicz**

1ª Lista de Exercícios / Implementação

Introdução

Nesta lista, vamos trabalhar em um labirinto (Pac-Maze) para exercitar o básico de alguns algoritmos de busca em espaço de estados (capítulos 3 e 4 do livro-texto).

Enquanto no Pac-Man original o objetivo é coletar todas as pastilhas ao fugir dos fantasmas que o perseguem, em nosso Pac-Maze o objetivo é apenas levar o Pac-Man até a única pastilha do cenário – sem fantasmas para atrapalhar. O cenário é um mundo bidimensional, representado por uma matriz de caracteres. A pastilha é representada por `0` (zero), uma parede por `#`, e um espaço vazio por `-`.

Para o sistema de coordenadas, usaremos a notação de matriz em um programa. A primeira coordenada é a linha e a segunda é a coluna. A origem (0,0) é o caractere superior esquerdo. No exemplo abaixo, o mundo possui 7 linhas e 14 colunas. A pastilha está em (1,6) - 2ª linha, 7ª coluna.

```
Origem (0,0) -> ##############
                #-----0--#---#
                #######-##-#-#
                #----------#-#
                #-##########-#
                #------------#
                ##############
```

Fig. 1 – Instância do Pac-Maze

Modelagem

O estado no Pac-Maze é simplesmente a localização do Pac-Man. Por exemplo, na Fig. 1 o objetivo é o estado (1,6).

Um ação no Pac-Maze corresponde às 4 direções de movimento do Pac-Man: acima, abaixo, esquerda e direita. Supondo que o Pac-Man comece na posição (5,1) da Fig. 1, as ações possíveis são: direita e acima. Ao aplicarmos a ação 'direita' no estado inicial (5,1), alcançamos o estado (5,2), representado na Fig. 2 (o Pac-Man está representado como `P`):

```
##############
#-----0--#---#
#######-##-#-#
#----------#-#
#-##########-#
#-P----------#
##############
```

Fig. 2 – Situação do Pac-Maze ao executarmos a ação 'direita' partindo de (5,1) na Fig. 1

Arquivo de entrada

Para os exercícios, você deverá ler o mundo do Pac-Maze. O arquivo de entrada tem em sua 1ª linha o tamanho do mundo: `m` linhas e `n` colunas, separados por espaço. As próximas `m` linhas contém `n` caracteres cada (além da quebra de linha). Os `n` caracteres são parede, pastilha ou espaço vazio (de acordo com a representação definida). Por exemplo, o conteúdo de um arquivo com o mundo da Fig. 1 é mostrado abaixo.

```
7 14
##############
#-----0--#---#
#######-##-#-#
#----------#-#
#-##########-#
#------------#
##############
```

Fig. 3 - Arquivo de entrada para o exemplo da Fig. 1

Exercício 1)

Para uma dada instância do Pac-Maze, a função `sucessor(estado)` recebe um estado e retorna uma lista de pares (ação;estado atingido) para cada ação que pode ser realizada no estado recebido como parâmetro. O par contém a ação que pode ser realizada no estado que foi recebido como parâmetro e o estado atingido ao se realizar aquela ação. Por exemplo, para o estado (5,1) no mundo da Fig. 1, `sucessor(5,1)` deve retornar uma lista com 2 pares (ação;estado) – observe que ação e estados são separados por ponto e vírgula e as coordenadas do estado separadas por vírgula:

```
(direita;5,2) (acima;4,1)
```

Implemente a função `sucessor`.

Como seu código será avaliado:

- Escreva um script `avalia_sucessor.sh` que execute o seu código. O `avalia_sucessor.sh` deve receber o caminho do arquivo de entrada com a descrição do mundo, as coordenadas de um estado e chamar o seu 'main'. Seu código deve imprimir na tela os pares (ação;estado) para cada estado atingido a partir do estado fornecido. Diferentes pares (ação;estado) devem estar separados por um espaço. Supondo que o arquivo descrito na Fig. 3 seja `pacmaze.txt`, a execução do seu script para consulta dos sucessores de (5,1) e a resposta esperada são da seguinte forma (a ordem dos pares não é importante):

```
avalia_sucessor.sh pacmaze.txt 5 1
(direita;5,2) (acima;4,1)
```

Exercício 2)

A partir a função `sucessor`, você será capaz de construir a função `expande` e um grafo de busca a partir de um estado inicial. No grafo de busca, cada nó contém informações sobre o estado a que ele se refere, além de informações que irão auxiliar a busca por uma solução.

As arestas do grafo de busca são as ações e os atributos de cada nó serão:

- Estado: representação do estado ao qual este nó se refere (e.g.: (5,1))
- Pai: referência ao nó que precede este
- Ação: ação que foi aplicada ao nó pai para gerar este
- Custo do caminho (path cost): o custo do caminho a partir do estado inicial até este nó. No caso do Pac-Maze, cada ação (aresta do grafo) terá custo 1. Assim, o custo de um nó é o custo do pai + 1.
- (opcional) referências para os nós filhos.

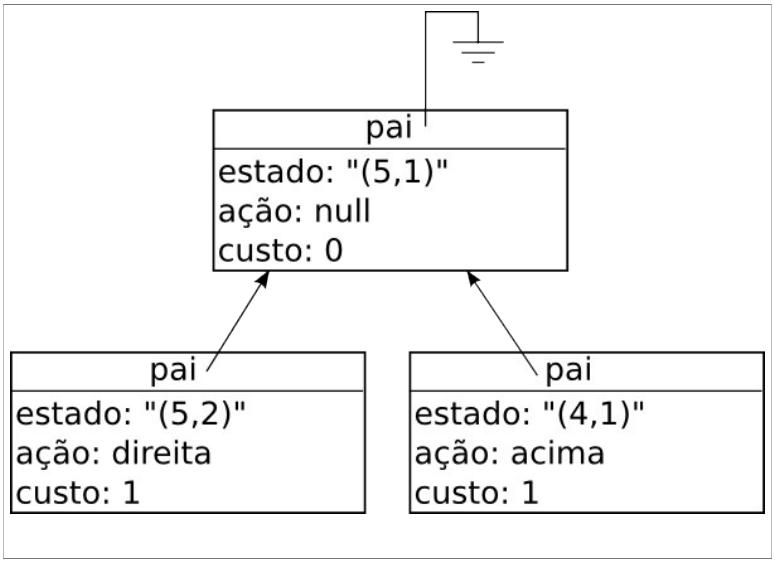

Fig. 4 – Representação de nós do Pac-Maze. Supondo que o estado inicial da Fig. 1 seja (5,1).

A função `expande` recebe um nó e retorna um conjunto de seus nós sucessores, usando a função `sucessor`. Por exemplo, ao passar o nó raiz da Fig. 3, os dois nós inferiores são retornados.

Implemente o nó do grafo de busca e a função `expande`.

Como seu código será avaliado:

- Escreva um script `avalia_expande.sh` que execute o seu código. O `avalia_expande.sh` vai receber o arquivo com a descrição do mundo, um estado para consulta e seu custo na linha de comando. Seu script deve chamar o seu 'main', que deve criar um nó com essas informações, chamar a função `expande` e imprimir o resultado esperado na tela. Seu código deve imprimir na tela as quádruplas (ação;estado;custo;pai.estado) para cada nó do conjunto retornado pela expansão. As quádruplas devem estar separadas por um espaço. Supondo que o arquivo descrito na Fig. 3 seja `pacmaze.txt`, a execução do seu script para o estado (5,1) com custo 3 e a resposta esperada são da seguinte forma (a ordem das quádruplas não é importante):

```
avalia_expande.sh pacmaze.txt 5 1 3
(direita;5,2;4;5,1) (acima;4,1;4;5,1)
```

(observe que: os itens de uma quádrupla são separados por ponto e vírgula, os estados são linha,coluna separados por vírgula, diferentes quádruplas são separadas por espaço, os estados dos pais de todos os nós expandidos são iguais e o custo dos filhos é o custo do pai + 1)

Exercício 3)

Com a função `expande`, será possível implementar o procedimento de busca em grafos. Neste procedimento, um nó do espaço de busca pertencerá ao conjunto `explorados` / fechado ou `fronteira` / aberto (fringe / open). O conjunto `explorados` armazena quais estados já foram expandidos e o tipo abstrato de dados para a `fronteira` armazena quais podem ser expandidos na sequência. A maneira como a `fronteira` é implementada dará origem aos diferentes algoritmos de busca (BFS, DFS, A*, etc.). Observe o pseudocódigo para busca geral em grafos (adaptado do livro-texto):

```
function graph_search(fronteira): //fronteira inicialmente vazia
    explorados ← {}
    fronteira.insere(new Node(estado_inicial))

    repeat:
        if fronteira.vazia():
            return "Sem solução"
        node ← fronteira.remove()
        if (node.estado é o objetivo):
            return solução(node)

        if node ∉ explorados:
            adiciona node a explorados
            fronteira.insere_lista(expande(node))
```

Observe que o nó com o estado inicial possui pai e ação nulos, já que ele é o primeiro. O avanço da busca depende de como a fronteira cresce, o que depende de qual nó será expandido. Além disso, o conjunto 'explorados' previne que nós sejam expandidos mais de uma vez (o que significaria revisitar um estado).

O tipo abstrato de dados de `fronteira` deve suportar as seguintes operações:

- `insere(Node n)`: adiciona o nó n à fronteira.
- `insere_lista(lista_de_nodes)`: adiciona todos os nós da lista à fronteira
- `remove()`: remove um nó da fronteira e o retorna.

A função `solução` recebe um nó e retorna a sequência de ações e estados que levam do estado inicial até o node recebido, ou seja, como o problema é resolvido.

O que você deve fazer:

- Implemente a busca em largura (BFS). Para isto, implemente a `fronteira` como uma fila.
- Implemente a busca em profundidade (DFS). Para isto, implemente a `fronteira` como uma pilha.
- Implemente o algoritmo A*. Para isto, a função `remove` da `fronteira` deve retornar o nó com menor custo `f(node) = g(node) + h(node)`. Nessa função de custo, `g(node)` é `node.custo` e `h(node)` é a distância heurística de node até um nó objetivo. Você escolherá a função de distância heurística.

Como seu código será avaliado:

- Escreva os scripts `avalia_bfs.sh, avalia_dfs.sh, avalia_astar.sh` que execute o código de cada um dos algoritmos de busca. Os scripts vão receber o arquivo de entrada com a descrição do mundo e as coordenadas do ponto de partida e chamar o seu programa, que vai executar o algoritmo de busca a partir do nó com a localização inicial do Pac-Man e imprimir o resultado na tela. Seu código deve imprimir na tela sequência de ações que levam do estado inicial para o estado objetivo (o local da pastilha). As ações devem estar separadas por um espaço. Por exemplo, supondo que o arquivo descrito na Fig. 3 seja `pacmaze.txt`, e o ponto de partida seja (5,1) a execução de um desses scripts e a resposta esperada são da seguinte forma:

```
avalia_bfs.sh pacmaze.txt 5 1
acima acima direita direita direita direita direita direita acima
acima esquerda
```

Exercício 4

Além dos programas, apresente em um `.pdf` os seguintes itens:

- As soluções do Exercício 3 encontradas para os arquivos `pacmaze-01-tiny.txt` e `pacmaze-02-small.txt` do `kit_ex1.zip`. Sugestões de estados iniciais: (5,1) e (1,1), respectivamente.
- Análise quantitativa comparando os algoritmos com relação ao número de estados expandidos, custo da solução e tempo de execução para TODOS os arquivos de entrada do `kit_ex1.zip`. **Apresente tabelas e/ou gráficos comparativos.**
- **Responda e explique:**
  - Sua heurística para o A* é admissível? Ela é consistente?
  - A busca em largura e/ou a busca em profundidade apresentam sempre soluções ótimas para o Pac-Maze?

Finalizando, você deve entregar um arquivo `.zip` contendo:

- Um script `avalia_sucessor.sh`, de acordo com as instruções do Exercício 1;
- Um script `avalia_expande.sh`, de acordo com as instruções do Exercício 2;
- Os scripts `avalia_bfs.sh, avalia_dfs.sh, avalia_astar.sh`;
- Um `.pdf` com as respostas do Exercício 4;
- Todos os arquivos do seu código-fonte usado na implementação.

ATENÇÃO: os shell scripts (`.sh`) e o `.pdf` devem se localizar na raiz do seu arquivo `.zip`! Isto é, o conteúdo do seu arquivo `.zip` deverá ser o seguinte:

```
compila.sh   (se você estiver usando uma linguagem compilada)
avalia_sucessor.sh
avalia_expande.sh
avalia_bfs.sh
avalia_dfs.sh
avalia_astar.sh
respostas.pdf
[arquivos do código fonte] <<pode criar subdiretórios se necessário
```

Conteúdo do arquivo `.zip` a ser enviado.

Observações finais

- Para o A*, as heurísticas garantem que menos nós do espaço de busca são expandidos, comparados ao BFS e DFS. Mesmo assim, a forma como a fronteira é implementada tem impacto direto no desempenho (tempo de execução) do A*. Uma implementação direta, com uma lista linear, pode demorar muito mais que a busca em largura e/ou em profundidade. Portanto, é aconselhável usar uma estrutura de dados mais inteligente.
- As implementações podem ser feitas na sua linguagem de programação favorita. No entanto, certifique-se que seu código funciona em uma máquina GNU/Linux (por exemplo: `tigre.dcc.ufmg.br`). Não imprima NADA ALÉM do que foi pedido, pois isso resultará em erro durante a correção. Em cada exercício, além do programa, você deverá escrever um shell script (.sh) simples que execute o seu programa com a entrada desejada. Caso use uma linguagem de programação compilada, escreva um script `compila.sh` que realize a compilação de todo o seu código. Para instruções sobre a escrita dos shell scripts, consulte o arquivo `auxiliar_shellscript.pdf`.
- Se você não usa GNU/Linux, tome cuidado com a codificação dos caracteres, em especial as quebras de linha. É realmente importante testar seu código em ambiente GNU/Linux, que é onde ele será corrigido. Se você não tem acesso às máquinas GNU/Linux do DCC, use uma máquina virtual para testar o seu programa. Uma lista de máquinas virtuais com Ubuntu pré-instalado está disponível em: http://www.osboxes.org/ubuntu/.