Segundo Trabalho de Inteligância Artificial (Busca Heurística em Grafos *OU*) (Prof. Alexandre Direne - 2013/2)

Atenção:

Este trabalho é obrigatório e deverá ser entregue, impreterivelmente, até o dia 06 de novembro 2013 (quarta-feira). A solução é individual e deverá ser arquivada no diretório "~alexd/IA/" onde o nome do arquivo terá como prefixo o seu nome-de-usuário no sistema do laboratório e, como extensão, ".pl" para indicar que seu conteúdo possui um programa em Prolog. Assim, por exemplo, se o seu nome de usuário no sistema fosse "grs09" então o nome o arquivo seria "grs09.pl" (dentro do diretório "~alexd/IA/"). Não se esqueça de proteger completamente o arquivo criado, de maneira a permitir a leitura do mesmo apenas por você! Isso pode ser feito aplicando `chmod og-rwx grs09.pl` antes de efetuar a cópia com a preservação das permissões (`cp -p grs09.pl ~alexd/IA/`). Não se preocupe com as permissões do professor que irá corrigir o trabalho. A correção dos trabalhos será parcialmente automatizada, sendo assim, é importante que todos os arquivos com as soluções individuais estejam no diretório citado acima, dentro do prazo estipulado. Não será permitida a entrega do arquivo por e-mail.

Enunciado:

Implementar em Prolog um predicado binário chamado `caixeiro` o qual realiza o trabalho de qualquer algoritmo de busca heurística para resolver o problema tradicional do Caixeiro Viajante. Atenção pois a quantidade de cidades pode ser grande (mais de 50), logo a solução exigida aqui não é minimalista em termos de custo do ciclo Hamiltoniano. Para ilustrar a aplicação do predicado `caixeiro`, deve bastar a carga de seu código fonte e a execução da seguinte consulta:

```
?- caixeiro( 2 , [[1 , 2 , 3539]] ).
Ciclo = [2, 1, 2]
Custo = 7078
Tempo de execucao = 0.01 segundo(s)
Produto = 70.78
yes
```

O primeiro termo de `caixeiro` representa a quantidade de cidades do mapa. A formação dos nomes das cidades segue a designação puramente numérica: $1, 2, \ldots, k$. Com isso, no exemplo de 50 (cinquenta) cidades, teríamos $k = 50$. O segundo termo de `caixeiro` representa a tabela de distâncias de todas as estradas entre as cidades. Lembre-se que no problema tradicional do Caixeiro Viajante o grafo de estradas entre cidades é `completo`. O predicado deverá ser adaptado para imprimir os três seguintes itens:

1. o ciclo Hamiltoniano encontrado;
2. o custo desse ciclo;
3. o tempo de execução da busca (ver detalhes do cálculo abaixo e siga-os exatamente assim);
4. o produto entre o custo e o tempo.

Posteriormente à entrega do trabalho, uma pequena comparação entre as soluções dos alunos será feita. Ganhará a comparação a solução que apresentar o menor `produto`. A instância de problema da competição terá entre 50 e 60 cidades. O fator tempo será considerado com valor mínimo de 8 (oito) segundos no cálculo do produto, mesmo se a execução do predicado `caixeiro` terminar antes. As bases de teste para 5, 15, 50, ..., 60 cidades podem ser encontradas em:

```
http://www.inf.ufpr.br/alexd/ia/teste5.pl
http://www.inf.ufpr.br/alexd/ia/teste15.pl
http://www.inf.ufpr.br/alexd/ia/teste50.pl
http://www.inf.ufpr.br/alexd/ia/teste51.pl
http://www.inf.ufpr.br/alexd/ia/teste52.pl
          ...
http://www.inf.ufpr.br/alexd/ia/teste60.pl
```

Para medir o consumo de tempo, veja o seguinte código aplicado ao cálculo do fatorial de um número aplicado a 20000. Veja os exemplos de uso tanto para o SWI-Prolog como para o POPLOG-Prolog e aplique no corpo do seu predicado principal `caixeiro`, de acordo com a sua opção de compilador.

Em SWI-Prolog, use o predicado `get_time` para medir o tempo. Veja o exemplo:

```
fat(0,1) :-
  !.
fat(X,Y) :-
  Z is X-1,
  fat(Z,Fz),
  Y is X*Fz.

tempo :-
  get_time(Inicio),
  fat(20000, _ ),
  get_time(Fim),
  Total is Fim - Inicio,
  write('Tempo = '),
  write(Total),
  write(' segundo(s)\n'),
  !.
```

Teste o exemplo acima assim:

```
?- tempo.
Tempo = 2.9560537338256836 segundo(s)
true.
```

Em POPLOG-Prolog, use uma chamada ao `systime` por meio do predicado `apply` e siga exatamente o mesmo modelo no cabeçalho no seu código fonte. Veja o exemplo:

```
:- prolog_language('pop11').
false -> popmemlim;
false -> pop_prolog_lim;
10e7 -> pop_callstack_lim;
true -> popdprecision;
12 -> pop_pr_places;
:- prolog_language('prolog').

fat(0,1) :-
  !.
fat(X,Y) :-
  Z is X-1,
  fat(Z,Fz),
  Y is X*Fz.

tempo :-
  Inicio is apply(valof(systime)),
  fat(20000, _ ),
  Fim is apply(valof(systime)),
  Total is (Fim - Inicio)/100.0,
  write('Tempo = '),
  write(Total),
  write(' segundo(s)\n'),
  !.
```

Teste o exemplo acima assim:

```
?- tempo.
Tempo = 0.41 segundo(s)
yes
```

Dicas de Prolog:

DICA 1: Sobre a geração aleatória de valores Reais entre 0 e 1 em SWI-Prolog, veja o exemplo

```
?- random(X).
X = 0.23319303367401667.

?- R is random(60) + 1.
R = 14.
```

Em POPLOG-Prolog, a geração aleatória de valores Inteiros entre 1 e um limite superior pode ser obtida assim:

```
?- Qtd_cidades is 50, prolog_eval(random(Qtd_cidades), Posicao).
Qtd_cidades = 50
Posicao = 15 ?
yes
```

DICA 2: A função exponencial em SWI-Prolog tendo o número Neperiano pode ser obtida assim (cuidaddo: o argumento da função `exp` tem que ser um número Real):

```
?- X is exp(1.0).
X = 2.718281828459045.
```

A função exponencial em POPLOG-Prolog tendo o número Neperiano pode ser obtida assim:

```
?- prolog_eval(exp(1.0), Resultado).
Resultado = 2.718281828459 ?
yes
```

Obsevações finais:

- Pode ser bem interessante conhecer os detalhes da buaca por Têmpera Simulada (*simulated annealing*) apresentados em mais de um livro de Inteligência Artificial (cuidado com a diferença entre os conceitos de *máximo* e *mínimo* local), assim como em páginas Web de boa qualidade, tal como:

  `http://en.wikipedia.org/wiki/Simulated_annealing`

- Não gaste seu tempo implementando qualquer mecanismo (*e.g.*, tabela hash ou qualquer estrutura de árvore) para acelerar da busca na tebela de custos unitários das estradas entre duplas de cidades;
- Não troque o nome do predicado `caixeiro` pois isto dificultará a correção do trabalho;
- Use apenas os compiladores SWI-Prolog ou POPLOG-Prolog.