Logic and Knowledge Representation
Exercise 3

1. SWI-Prolog

SWI-Prolog offers a comprehensive free Prolog environment. Since its start in 1987, SWI-Prolog development has been driven by the needs of real world applications. SWI-Prolog is widely used in research and education as well as commercial applications.

Download at *http://it.tdt.edu.vn/~dhphuc/teaching/artificial-intelligence/swipl-w64-741.exe*

2. Sample program

Following is an example of a *family tree* problem.

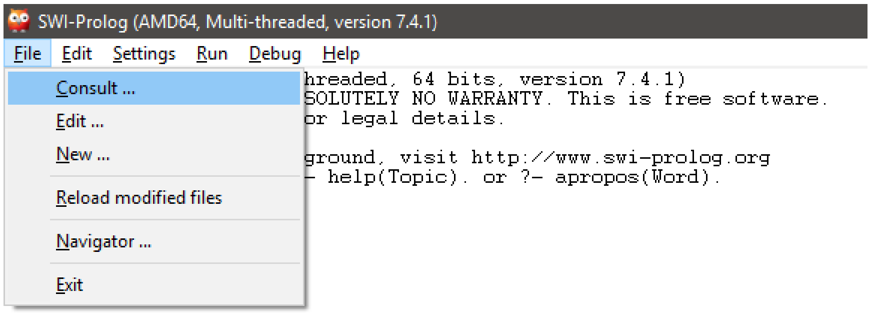
```
% AI-Prolog-FamilyTreeExample
male(tom).
male(bob).
male(jim).
female(liz).
female(pat).
female(ann).
female(pam).

% pam is parent of bob
parent(pam,bob).
parent(tom,bob).
parent(tom,liz).
parent(bob,ann).
parent(bob,pat).
parent(pat,jim).

% X is father of Y
father(X,Y):- parent(X,Y),male(X).
offspring(X,Y):- parent(Y,X).
pred(X,Y):-parent(X,Y).
pred(X,Y):-parent(X,Z),pred(Z,Y).
same(X,Y):- X=Y.
diff(X,Y):- not(same(X,Y)).
```

First, copy and paste the above code to a new text file, named it by *family_tree.pl*.

Second, open SWI-Prolog and compile the KB file, as present in **Figure 1**.



```
hreaded, 64 bits, version 7.4.1)
SOLUTELY NO WARRANTY. This is free software.
or legal details.

ground, visit http://www.swi-prolog.org
- help(Topic). or ?- apropos(Word).
```

Figure 1 Consult a KB in SWI-Prolog

Then, type the following commands to get inference results from KB.

- `parent(tom,bob).`
- `mother(tom,bob).`
- `parent(Y,ann).`

Try to write inference rules to find out the *mother*, *sister*, *daughter*, and *son* relationship.

3. The A* algorithm in Prolog

Heuristic search uses a heuristic function to help guide the search. When a node is expanded, each of its children is evaluated using a search function. Each child is placed into a list of nodes – the so-called open list – in order determined by the search function evaluation (smaller values first). The heuristic function estimates how much work must be done to reach a goal from the node in question. Typically, the search function *f* is expressed as

$$f(n) = g(n) + h(n)$$

where $g(n)$ represents the (computed, actual) cost of getting to the node *n* along the current path to it, and *h* is the heuristic function. Thus, $f(n)$ estimates the cost or effort to successfully get from start to goal by going through node *n* (along some paths).

A simple pseudocode from which the Prolog program will be developed:

- Start with the start node, place it in the (previously empty) list **open**.
- Let *n* be the first node on **open**. Remove *n* from **open**. Fail if **open** is empty.
- If *n* is the goal, then a solution has been found. (One could stop here.)
- Expand *n*, obtaining all of its children, and evaluate $f(-)$ for each of them. Insert each of these children into **open**, maintaining order where the smallest $f(-)$ values come first.
- Repeat from step 2.

A common cost function *g*(-) is path length. The cost of getting from the start node to a current node is the length of relevant path. This can be computed incrementally.

It is important to realize that this kind of search can follow a contiguous path for a while, until some previously unchosen node *n* has the current smallest *f*(-) value, in which case this node *n* is expanded, and its children considered.

Let's assume that **State** refers to some description of the **State** of a search. A **Node** in the search space (or graph) needs to record the **State**, the Depth (or path length from the start), the value of *f*(-) for the **Node F**, and a list of the ancestors **A** of this **Node**. We will use the following Prolog term structure for a **Node**.

Node = State#Depth#F#A

When **Node** is expanded to find its children, it performs those tasks below:

- The state of each child will be computed as a move from **State**,
- Each of these children will have depth **Depth + 1**,
- The *f*(-) value of each child will be calculated,
- The ancestor list of a child will be the Prolog list term **[Node|A]**.

Read more: *http://it.tdt.edu.vn/~dhphuc/teaching/artificial-intelligence/prolog_tutorial/5_1.html*.

4. Apply A* in 8-Puzzle

To represent these puzzle "states" we will use a Prolog term representation employing '/' as a separator. The positions of the tiles are listed (separated by '/') from top to bottom, and from left to right. Use "0" to represent the empty tile (space). For example, the goal is **goal(1/2/3/8/0/4/7/6/5)**.

Read more: *http://it.tdt.edu.vn/~dhphuc/teaching/artificial-intelligence/prolog_tutorial/5_2.html*.

5. Question

Question 1. Implement the A* algorithm and apply to 8-Puzzle problem.