[CSE537] The Pacman using Prolog

REPORT for TEAM PROJECT #3

by Oleksii Starov, Hyungjoon Koo

Nov. 4, 2013

A. General Information

(1) Team-up (Group 41): Oleksii Starov (ostarov@cs.stonybrook.edu), Hyungjoon Koo (hykoo@cs.stonybrook.edu)

(2) Project information: Pacman using Prolog (PJT#3), http://www.cs.sunysb.edu/~ram/cse537/project03-2013.html (Due date: Nov. 4, 2013)

(3) Implementation: BFS, DFS, and A* in Prolog

B. Implementation Note

(1) **Common implementation to call XFS from python (Basic Step)**:

```
from game import Directions
from spade import pyxf
import time

# Creates an instance and makes a query to XSB using SPADE interface
myXsb = pyxf.xsb('C:/XSB/config/x86-pc-windows/bin/xsb.exe')
myXsb.load('D:/Tmp/maze.P')
myXsb.load('D:/Tmp/bfs.P')

time.sleep(2)
result = myXsb.query("bfs(start,finish,Direction).")

finalDirections = []
res = []

# Choose the shortest path from all results available.
if isinstance(result,bool):
   print "Catches XSB Problem!"
else:
   cnt = 99999
   for dict in result:
      if len(dict['Direction']) < cnt:
          cnt = len(dict['Direction'])
          res = dict['Direction']
          res = res.split(",")
          break

# Perform final step to return real directions to Pacman
   for direction in res:
     if direction=="s":
        finalDirections.append(Directions.SOUTH)
     if direction=="w":
        finalDirections.append(Directions.WEST)
     if direction=="n":
        finalDirections.append(Directions.NORTH)
     if direction=="e":
        finalDirections.append(Directions.EAST)
return finalDirections
```

First off, we were able to create an instance and made a query to XSB through SPADE interface. But the XSB behave quite unreliable from time to time. So we decided to handle this (1) by inserting the exception handling code to catch an error, and (2) by giving a couple of seconds while running a query. This code will choose the shortest path available and returns it to Pacman after appropriate conversion.

(2) **Writing the "maze.P" from wall information during running time**

```
# Create the "FACTS" from wall information in "maze.P" during running time
wallInfo={}

wallWidth = self.walls.width
wallHeight = self.walls.height

mazeFile = file('D:/Tmp/maze.P','w')
facts = []
cellCtr=1

# Initialize wall information
# True if there exists a wall or False otherwise
for i in range(0,wallWidth):
    for j in range(0,wallHeight):
        wallInfo[i,j] = self.walls[i][j]

# Check all cells other than outer perimeters and store the "ground facts" to the list.
for j in range(1,wallHeight-1):
    for i in range(1,wallWidth-1):
        # Check if the current position is a start state.
        if (i,j) == self.startState:
            groundStart = "connect(start,cell" + str(cellCtr) + ",na)."
            facts.append(groundStart)
        # Check if the current position is a goal state.
        if (i,j) == self.goal:
            groundGoal = "connect(cell" + str(cellCtr) + ",finish,na)."
            facts.append(groundGoal)
        # Look at the upper position and determine whether the cell above is connected or not.
        if wallInfo[i,j] == False and wallInfo[i,j+1] == False :
            groundCurrentUp = "connect(cell" + str(cellCtr) + ",cell" + str(cellCtr+wallWidth-2) + ",n)."
            groundUpCurrent = "connect(cell" + str(cellCtr+wallWidth-2) + ",cell" + str(cellCtr) + ",s)."
            facts.append(groundCurrentUp)
            facts.append(groundUpCurrent)
        # Look at the right position and determine whether the cell on the right is connected or not.
        if wallInfo[i,j] == False and wallInfo[i+1,j] == False :
            groundCurrentRight = "connect(cell" + str(cellCtr) + ",cell" + str(cellCtr+1) + ",e)."
            groundRightCurrent = "connect(cell" + str(cellCtr+1) + ",cell" + str(cellCtr) + ",w)."
            facts.append(groundCurrentRight)
            facts.append(groundRightCurrent)
        cellCtr= cellCtr+1

# Write the facts into the file
mazeFile.write('\n'.join(facts))
```

After storing the wall information, we checked if each cell is connected to adjacent cells. If a cell is a start state or a goal state, then we mark it. For example, since *cell1* is a goal state and *cell25* is an initial state, they can be represented as *connect(cell1, finish, na)* and *connect(start, cell25, na)* respectively. The last element means the direction.

Here is a sample output of ground facts when running *tinySize* map.

```
connect(cell1,finish,na).
connect(cell1,cell2,e).
connect(cell2,cell1,w).
connect(cell2,cell7,n).
connect(cell7,cell2,s).
connect(cell7,cell12,n).
connect(cell12,cell7,s).
connect(cell7,cell8,e).
connect(cell8,cell7,w).
connect(cell8,cell9,e).
connect(cell9,cell8,w).
connect(cell9,cell14,n).
connect(cell14,cell9,s).
connect(cell11,cell16,n).
connect(cell16,cell11,s).
connect(cell11,cell12,e).
connect(cell12,cell11,w).
connect(cell14,cell15,e).
connect(cell15,cell14,w).
connect(cell15,cell20,n).
connect(cell20,cell15,s).
connect(cell16,cell21,n).
connect(cell21,cell16,s).
connect(cell20,cell25,n).
connect(cell25,cell20,s).
connect(cell21,cell22,e).
connect(cell22,cell21,w).
connect(cell22,cell23,e).
connect(cell23,cell22,w).
connect(cell23,cell24,e).
connect(cell24,cell23,w).
connect(cell24,cell25,e).
connect(cell25,cell24,w).
connect(start,cell25,na).
```

(3) **Prolog codes**

We have several helper predicates for search: member, append, and reverse.

DFS.P

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%        PART I: Helper Function         %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Check if X is a member of the list
% Usage: member(element,[list]).
% ?- member(a,[d,c,a]).
% true.
member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%          PART II: Main Function        %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Usage: dfs(goal, initial, Solution)
% ?- dfs(finish,[start],Direction).
dfs(Goal, [Goal|_], []).
dfs(Goal, [Current|Past], [Dir|PrevDir]) :-
        connect(Current,Next,Dir),
        not(member(Next,Past)),
        dfs(Goal,[Next,Current|Past], PrevDir).
```

BFS.P

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%        PART I: Helper Functions        %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Check if X is a member of the list
% Usage: member(element,[list]).
% ?- member(a,[d,c,a]).
% true.
member(X,[X|List]).
member(X,[Y|List]) :- member(X,List).

% Append the list A into the list B
% Usage: append([listA],[listB],Result).
% ?- append([a,b,c],[d,e],Ans)
% Ans = [a, b, c, d, e].
append([],List,List).
append([Head|Tail],List2,[Head|Result]):- append(Tail,List2,Result).

% Reverse all elements in the Inputlist to the Outputlist
% Usage: reverse([Inputlist],Outputlist).
% ?- reverse([a,b,d,e],Ans).
% Ans = [e, d, b, a].
reverse(Inputlist,Outputlist):-reverseHelper(Inputlist,[],Outputlist).
reverseHelper([],Outputlist,Outputlist).
reverseHelper([Head|Tail],List1,List2):-
reverseHelper(Tail,[Head|List1],List2).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%          PART II: Main Function        %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Includes cycle detection and returns the first optimal path found
% Needs maze.P file containing ground facts
% i.e. connect(cell#,cell#,direction).
% Usage: bfs(initial, goal, Solution)
% ?- bfs(start, finishl, Direction)

bfs(X,Y,P) :- bfsHelper(Y,[n(X,[])],[n(X,[])],R), reverse(R,P).
bfsHelper(Y,[n(Y,P)|_],_,P).
bfsHelper(Y,[n(S,P1)|Ns],C,P) :-
        findall(n(S1,[A|P1]),(connect(S,S1,A),
        \+ member(n(S1,_),C)),Es),
        append(Ns,Es,O),
        append(C,Es,C1),
        bfsHelper(Y,O,C1,P).
```

Here are comments about how *BFS.P* works.

```
dfs(Goal, [Goal|_], []).                          % Base case which the goal is found
dfs(Goal, [Current|Past], [Dir|PrevDir]) :-
       connect(Current,Next,Dir),                 % Check if the Current node is connected to the Next
       not(member(Next,Past)),                    % If the node is not explored
       dfs(Goal,[Next,Current|Past], PrevDir).    % Then recursively explore the successors
```

Here are comments about how *DFS.P* works.

```
bfs(X,Y,P) :- bfsHelper(Y,[n(X,[])],[n(X,[])],R),   % Initially enter start, finish
       reverse(R,P).                                 % reverse the solution from start to end
bfsHelper(Y,[n(Y,P)|_],_,P).                         % base case
bfsHelper(Y,[n(S,P1)|Ns],C,P) :-
       findall(n(S1,[A|P1]),(connect(S,S1,A),        % Find all successors and check if they are connected
       \+ member(n(S1,_),C)),Es),                    % If those are not explored
       append(Ns,Es,O),                              % Append all found sucessors into queue.
       append(C,Es,C1),                              % Mark them as explored nodes in another list
       bfsHelper(Y,O,C1,P).                          % Recursively call bfsHelper
```

We tried to do A* search until the last minute, but there seems to have unresolved problem.

The following python code is for A* search with heuristic values and weight for cells, which does not achieve the final goal.

```
# Initialize wall information and heuristics
   # True if there exists a wall or False otherwise
   for i in range(0,wallWidth):
      for j in range(0,wallHeight):
         # Create clauses for heuristic value for each cell
         if i > 0 and i < wallWidth-1 and j > 0 and j < wallHeight-1:
            xy1 = (i,j)
            xy2 = goal
            heuristicValue = abs(xy1[0] - xy2[0]) + abs(xy1[1] - xy2[1])
            if (i,j) == self.startState:
               groundHeuristic = "heuristic(start," + str(heuristicValue) + ")."
               facts.append(groundHeuristic)
            elif (i,j) == self.goal:
               groundHeuristic = "heuristic(finish,0)."
               facts.append(groundHeuristic)
            else:
               cellNum = i+(j-1)*(wallWidth-2)
               groundHeuristic = "heuristic(cell" + str(cellNum) + "," + str(heuristicValue) + ")."
               facts.append(groundHeuristic)
         wallInfo[i,j] = self.walls[i][j]

   # Check all cells other than outer perimeters and store the "ground facts" to the list.
   for j in range(1,wallHeight-1):
      for i in range(1,wallWidth-1):
         # Check if the current position is a start state.
         if (i,j) == self.startState:
            groundStart = "connect(start,cell" + str(cellCtr) + ",na)."
            groundStartCost = "weight(start,cell" + str(cellCtr) + ",0)."
            facts.append(groundStart)
            facts.append(groundStartCost)
         # Check if the current position is a goal state.
         if (i,j) == self.goal:
            groundGoal = "connect(cell" + str(cellCtr) + ",finish,na)."
            groundGoalCost = "weight(cell" + str(cellCtr) + ",finish,0)."
            facts.append(groundGoal)
            facts.append(groundGoalCost)
         # Look at the upper position and determine whether the cell above is connected or not.
         if wallInfo[i,j] == False and wallInfo[i,j+1] == False :
            cost = self.costFn((i,j+1))
            groundCurrentUp = "connect(cell" + str(cellCtr) + ",cell" + str(cellCtr+wallWidth-2) + ",n)."
            groundUpCurrent = "connect(cell" + str(cellCtr+wallWidth-2) + ",cell" + str(cellCtr) + ",s)."
            currentCost1 = "weight(cell" + str(cellCtr) + ",cell" + str(cellCtr+wallWidth-2) + "," + str(cost) + ")."
            currentCost2 = "weight(cell" + str(cellCtr+wallWidth-2) + ",cell" + str(cellCtr) + ","+ str(cost) + ")."
            facts.append(groundCurrentUp)
            facts.append(groundUpCurrent)
            facts.append(currentCost1)
            facts.append(currentCost2)
```

```
        # Look at the right position and determine whether the cell on the right is connected or not.
        if wallInfo[i,j] == False and wallInfo[i+1,j] == False :
           cost = self.costFn((i+1,j))
           groundCurrentRight = "connect(cell" + str(cellCtr) + ",cell" + str(cellCtr+1) + ",e)."
           groundRightCurrent = "connect(cell" + str(cellCtr+1) + ",cell" + str(cellCtr) + ",w)."
           currentCost1 = "weight(cell" + str(cellCtr) + ",cell" + str(cellCtr+1) + "," + str(cost) + ")."
           currentCost2 = "weight(cell" + str(cellCtr+1) + ",cell" + str(cellCtr) + ","+ str(cost) + ")."
           facts.append(groundCurrentRight)
           facts.append(groundRightCurrent)
           facts.append(currentCost1)
           facts.append(currentCost2)
        cellCtr= cellCtr+1
```

The following Prolog code is for A* search with comments in details.

```
:- import append/3, reverse/2, length/2, between/3, member/2 from basics.

% For logging, also with SPY
printlist([]).
printlist([X|List]) :- write(X),nl,printlist(List).

% Key-Value Table for current best distances per a vertex
% For check for improved distance

getByKey([], K, 9999999).
getByKey([n(K,_,_,G)|Pairs], K, G).
getByKey([n(Key,_,_,_)|Pairs], K, Res) :- getByKey(Pairs, K, Res).

% Permutation generator
perm([A|B],L):- length(L,N), between(0,N,I),length(X,I),
  append(X,[A],Y), append(Y,Z,L),
  append(X,Z,M), perm(B,M).
perm([],[]).

% Sort by predicate
sort(O,L,S) :- perm(L,S), ordered(O,S).
ordered(_,[_]).
ordered(O,[X,Y|Ys]) :- G =.. [O,X,Y], call(G), ordered(O,[Y|Ys]).

% HERE THE FUN BEGINS!
astar(X,Y,P) :-
astarHelper(Y,[n(X,[],[],0)],R),
reverse(R,P).

% My comparator of nodes - by distance value
g_less(n(A,Pa,PPa,Ga), n(B,Pb,PPb,Gb)) :- Ga < Gb.

% Sorting instead of a priority queue

astarHelper(Y,[n(Y,P,PP,_)|_],P).
```

```
astarHelper(Y,[n(S,P1,PP1,G)|Ns],P) :-
writeln(S),
writeln(G),
findall(n(S1,[A|P1],[S|PP1],G1),
(connect(S,S1,A),
weight(S,S1,C), heuristic(S1,H), G1 is G + C + H),
Es),
append(Ns,Es,O1),
sort(g_less,O1,O),
printlist(O),
astarHelper(Y,O,P).

% DAG EXAMPLE:

%Ans = [na,e,e,e,na]
% S, A, D, G from slide 13, lecture Search Supplement

connect(start,cell1,na).
connect(cell1,cell2,e).
connect(cell1,cell3,n).
connect(cell2,cell4,s).
connect(cell2,cell5,e).
connect(cell5,cell3,w).
connect(cell5,cell6,e).
connect(cell3,cell5,s).
connect(cell3,cell6,n).
connect(cell6,finish,na).

weight(start,cell1,0).
weight(cell1,cell2,2).
weight(cell1,cell3,5).
weight(cell2,cell4,2).
weight(cell2,cell5,4).
weight(cell5,cell3,3).
weight(cell5,cell6,2).
weight(cell3,cell5,1).
weight(cell3,cell6,5).
weight(cell6,finish,0).

heuristic(start,0).
heuristic(cell1,0).
heuristic(cell2,2).
heuristic(cell3,3).
heuristic(cell4,1).
heuristic(cell5,1).
heuristic(cell6,0).
heuristic(finish,0).
```

C. The results

```
# python pacman.py -l [MazeSize] -p SearchAgent -a fn=[SearchKind] -z .5
```

Size of Maze	DFS	BFS
TinyMaze	[{'Direction': '[na,s,s,w,s,w,w,s,w,na]'}, {'Direction': '[na,w,w,w,w,s,s,e,s,s,w,na]'}]	[{'Direction': '[na,s,s,w,s,w,w,s,w,na]'}]
MediumMaze	[{'Direction': '[na,s,s,w,w,w,w,s,s,s,e,e,e,e,s,s,w,w,w,w,s,s,e,e,e,e,s,s,w,w,w,w,s,s,s,e,n,e,e,e,s,s,s,w,w,w, w,w,w,w,n,w,w,w,w,w,w,w,w,w,w,w,w,w,w,w,w,w,w,w,s,w,w,w,w,w,w,w,w,w,na]'}, {'Direction': '[na,s,s,w,w,w,w,s,s,s,e,e,e,e,s,s,w,w,w,w,s,s,e,e,e,e,s,s,w,w,w,w,s,s,s,e,n,e,e,e,s,s,s,w,w,w, w,w,w,w,n,n,n,n,n,n,n,n,w,w,w,w,w,w,w,n,n,w,n,w,w,w,w,w,s,s,e,e,e,s,s,w,w,w,w,w,s,s ,w,w,w,w,w,s,w,w,w,w,w,s,s,e,e,e,e,e,e,e,n,e,e,e,e,e,n,n,e,e,e,e,e,e,s,s,w,w,w,s,s,w,w, w,w,w,s,w,w,w,w,w,w,w,w,w,na]'}, {'Direction': '[na,s,s,w,w,w,w,s,s,e,e,e,e,s,s,w,w,w,w,s,s,e,e,e,e,s,s,w,w,w,w,s,s,s,e,n,e,e,e,s,s,s,w,w,w, w,w,w,w,n,n,n,n,n,n,n,n,w,w,w,w,w,w,w,n,n,w,n,w,w,w,w,w,s,s,e,e,e,s,s,w,w,w,w,w,s,s ,w,w,w,w,w,s,w,w,w,w,w,s,s,e,e,e,e,e,e,e,n,e,e,e,e,e,n,n,e,e,e,e,e,e,n,e,e,e,e,e,s,s,w,w,w, s,w,w,w,w,w,w,s,s,w,w,w,w,s,w,w,w,w,w,w,w,w,w,na]'}, {'Direction': '[na,s,s,w,w,w,w,s,s,e,e,e,e,s,s,w,w,w,w,s,s,e,e,e,e,s,s,w,w,w,w,s,s,s,e,n,e,e,e,s,s,s,w,w, w,w,w,w,n,n,n,n,n,n,n,n,w,w,w,w,w,w,w,n,n,w,n,w,w,w,w,w,s,s,e,e,e,s,s,w,w,w,w,w,s,s ,w,w,w,w,w,s,w,w,w,w,w,s,s,e,e,e,e,e,e,e,n,e,e,e,e,e,n,n,e,e,e,e,e,e,n,e,e,e,e,e,s,s,w,w, s,w,w,w,w,w,w,s,s,w,w,w,w,s,w,w,w,w,w,w,w,w,w,na]'}]	[{'Direction': '[na,w,w,w,w,w,w,w,w,w,s,s,e,e,s,s,s,w,w,w,n,w,w,w,w,s,s,s,e,e,e,e,e, e,e,s,s,s,s,s,s,s,w,w,w,w,w,w,w,w,w,w,w,w,w,w,w,s,w,w,w,w,w,w, w,w,w,na]'}]
BigMaze	[{'Direction': '[na,n,n,w,w,w,w,n,n,w,w,s,s,w,w,w,w,w,w,w,w,w,w,w,w,w,n,n,e,e,n,n,w,w,n,n,n,n,n,n ,e,e,e,e,e,e,s,s,e,e,n,n,e,e,e,e,n,n,e,e,s,s,e,e,n,n,n,n,n,n,e,e,e,e,e,e,s,s,e,e,n,n,e,e,e,e,n,n,e,e,s,s,e,e,n,n,n, s,w,w,w,w,s,s,s,s,s,s,w,w,s,s,s,s,w,w,n,n,w,w,w,w,w,w,w,w,w,w,w,w,n,n,e,e,n,n,n,n,n,n,e, e,e,e,e,e,n,n,n,n,n,n,n,n,w,w,w,w,w,w,s,s,w,w,w,s,s,s,s,e,e,s,s,w,w,w,w,w,w,w,w,s, s,s,s,s,s,s,s,s,s,e,e,s,s,s,s,w,w,s,s,s,s,e,e,s,s,w,w,s,s,s,s,w,w,s,s,na]'}]	[{'Direction': '[na,n,n,w,w,w,w,n,n,w,w,s,s,w,w,w,w,w,w,w,w,w,w,w,w,w,n,n,e,e,n,n,w,w,n,n,n,n,n,n,e,e,e, n,n,w,w,n,n,n,n,n,n,e,e,e,e,e,e,s,s,e,e,n,n,e,e,e,e,n,n,e,e,s,s,e,e,n,n,n, n,n,n,e,e,e,e,n,n,n,n,n,n,n,n,n,n,w,w,s,s,w,w,w,w,s,s,s,s,s,s,w,w,s,s,s,s ,w,w,n,n,w,w,w,w,w,w,w,w,w,w,w,n,n,e,e,n,n,n,n,n,n,e,e,e,e,e,e,n, n,n,n,n,n,n,n,w,w,w,w,w,w,s,s,w,w,w,w,s,s,s,s,e,e,s,s,w,w,w,w,w,w, w,w,w,s,s,s,s,s,s,s,s,s,e,e,s,s,s,s,w,w,s,s,s,s,e,e,s,s,w,w,s,s,s,s,w,w,s,s, na]'}]
TestMaze	[{'Direction': '[na,w,w,w,w,w,w,w,na]'}]	[{'Direction': '[na,w,w,w,w,w,w,w,na]'}]

The table above shows the results of depth first search (*dfs.P*) and breadth first search (*bfs.P*) at each maze respectively. The DFS returns a set of possible directions while BFS returns the optimal result.

D. References

http://www.cs.nott.ac.uk/~bsl/G52APT/

http://www.cs.nott.ac.uk/~bsl/G52APT/slides/10-Breadth-first-search.pdf

http://www.cs.nott.ac.uk/~bsl/G52APT/slides/08-Depth-first-search.pd

http://www.cs.nott.ac.uk/~bsl/G52APT/slides/11-Representing-states.pdf