```
% BEST-FIRST SEARCH ROUTINES FOR VIRTUAL WORLD
%(best_first_search.pl)
% This program incorporates greedy and A-Star, using
% either h(n) only for greedy, or f(n)=g(n)+h(n) for A*
% It uses a heuristic to estimate h(n) as the number
% of tiles to move vertically or horizontally (max of these)
% for the shortest distance estimate. It also calculates
% the actual cost as 10 for orthogonal moves and
% 14 for diagonals. The path is chosen as the best
% in A* and f(n)=h(n)+g(n), ie best = smallest f(n)
%
% Original code Copyright 2005 - Donald Nute
% Last modified by Donald Nute: 16/8/2005
% modifications and significant additions
% by Graham Winstanley, June, 2010
%
% This file includes a sample agent's map of a virtual world
% It isslightly changed from Nute's original map
% Please refer to the documents "Workshop 4" and
% "Developing best-first search" for a complete
% explanation of this code

% call with (e.g.)
%        find_path(a_star,door,(0,0),_).

:- dynamic [search_type/1, mygoal/1].

search([FirstNode|_],FirstNode,Goal_X,Goal_Y,CLOSED,OPEN) :-       % take the
first node
      solution(FirstNode).          % is it the goal tile?

search([FirstNode|RestOfNodes],Solution,Goal_X,Goal_Y,CLOSED,OPEN) :-   %
from the current tile
      generate_new_nodes(FirstNode,NewNodes,Goal_X,Goal_Y), % generate next
poss tiles
      insert_nodes(NewNodes,RestOfNodes,NextSetOfNodes),     % add to OPEN
      !,
      add_to_CLOSED(NextSetOfNodes,CLOSED,New_CLOSED),       % put the next
node (best) on CLOSED
      check_CLOSED(NextSetOfNodes,New_CLOSED,NextSetOfNodes_2),    % better
path on CLOSED?
      nl, write(' NextSetOfNodes (OPEN): ' - NextSetOfNodes_2),nl,ttyflush,
      nl, write(' CLOSED: ' - New_CLOSED),nl,ttyflush,
      search(NextSetOfNodes_2,Solution,Goal_X,Goal_Y,New_CLOSED,NextSetOfNode
s_2). % recurse with new list


% This caters for the case where there may be multiple paths
% to the same node. Use findall/3
/* check_CLOSED2([[2,0,2,(1,0),(0,0)], [3,0,3, (2,2),(1,1)]],
      [[4,0,4,(1,1),(0,0)], [3,0,3,(3,4),(4,2)],
[1.9,0,1.9,(1,0),(0,0)],[6,0,6,(6,0),(6,8)], [1.8,0,1.8,(1,0),(0,0)]], N).

there are 2 occurrences of [_,(1,0),(0,0)] in CLOSED, one with Cost=1.5 & the
other 1.8
*/
```

```
check_CLOSED([First|Rest],[New_addition|CLOSED],NextSetOfNodes_2):-     %
strip off the first (best?)
      CLOSED = [] -> NextSetOfNodes_2 = [First|Rest] ;                  %
empty list? do nothing
      (First=[A,B,C,D|Closed],                                    % gets the cost &
head of First
      findall(Best_path,
      (
      member([_,_,_,D|Closed],CLOSED,Pos),                  % is the node (B) in
CLOSED?
      mem(CLOSED,[Pos],Best_path)                           % gets the path at Pos
      ),
      Paths)),
      sort(Paths, Sorted_paths,[1]),                        % sort them on Cost
      Sorted_paths = [Best|_],                              % retrieve the best
path
      Best = [Fn,_,_|_],                                    % retrieve the cost of
best
      Fn<A ->                                               % is the cost smaller
on CLOSED?
      append([Best],Rest,NextSetOfNodes_2);                 % append as a list..
[First]
      NextSetOfNodes_2 = [First|Rest].


add_to_CLOSED([First|Rest],CLOSED,New_CLOSED):-             % strip off the first
      append([First],CLOSED,New_CLOSED).                    % append as a list..
[First]


% The method used to insert new nodes in the queue determines
% which kind of search is used. You should assert either
% doing_depth_first_search/0 or doing_breadth_first_search/0
% before beginning your search.

% For depth first search, put new nodes at front of queue.

insert_nodes(Set1,Set2,Set3) :-       % NewNodes, RestOfNodes, Next...
      search_type(depth_first),       % if it is DF, put NewNodes at front
      append(Set1,Set2,Set3).         % = a stack

% For breadth first search, put new nodes at back of queue.

insert_nodes(Set1,Set2,Set3) :-
      search_type(breadth_first),     % if it is BF, put NewNodes at back
      append(Set2,Set1,Set3).         % = a queue

% For greedy search, need to sort the queue best at front = smallest h(n).
% first append to complete the list & then do a sort. sort/3 takes
% a list, in this case [Fn,Gn,Hn,[X,Y, X,Y...]] and uses the third element
% of each path [3] as the sort key. Ascending order too

insert_nodes(Set1,Set2,Sorted):-
      search_type(greedy),      % if it is greedy, do sort on h(n)
      append(Set2,Set1,Set3),   % do something clever!
      sort(Set3,Sorted,[3]).    % sort on h(n) only for greedy
```

```
% For a_star search, need to sort the queue best at front = smallest f(n).
% first append to complete the list & then do a sort. sort/3 takes
% a list, in this case [Fn,Gn,Hn,[X,Y, X,Y...]] and uses the first element
% of each path [3] as the sort key. Ascending order too

insert_nodes(Set1,Set2,Sorted):-
      search_type(a_star),     % if it is greedy, do sort on h(n)
      append(Set2,Set1,Set3), % do something clever!
      sort(Set3,Sorted,[1]).  % msgbox(`Sorted path `,Sorted,16'00000031,_).


how_long(Goal,Duration,Error) :-
      time(0,Start),
      catch(Error,Goal,Return),
      time(0,Finish),
      Start = (SDays,SMilliseconds),
      Finish = (FDays,FMilliseconds),
      Duration is (86400000 * FDays) + FMilliseconds - (86400000 * SDays) -
SMilliseconds,
      (
       Return = 0,
       Message = `Goal succeeded.`
      ;
       error_message(Error,Message)
      ).

% Now we define a function specific to V-World that uses the
% generalized search algorithm.
%
% find_path(SearchType,Goal,Start,Path) searches for a list of
% coordinates (a Path) that leads from the agent's current
% location Start (of the form (X,Y)) to a location where
% an object of the sort specified by Goal is located. Goal
% might be tree or cross, for example. The kind of search
% peformed is determined by SearchType, which can be depth_first
% or breadth_first. To simplify programming, % the Path is built
% backwards and then reversed.
%
% The SearchType and Goal are asserted into clauses where they
% can be used by insert_nodes/3 and solution/1.

find_path(SearchType,Goal,Start,Path) :-
      retractall(search_type(_)),         % house keeping
      assert(search_type(SearchType)),
      retractall(mygoal(_)),
      assert(mygoal(Goal)),               % assert the given goal
      findall((X,Y,Goal),
            mymap(X,Y,Goal),
            Goals),
      get_closest(Goals, X_Goal,Y_Goal),
      search([[Fn,Gn,Hn,Start]],ReversedPath,X_Goal,Y_Goal,[],[]),      %
call the top-level search
      reverse(ReversedPath,Path),         % now reverse it - start-to-goal
path
      nl,write('Final Path = ' - Path),nl,ttyflush,! . % print it - final
path
```

```
% A path satisfies a Goal if the first (eventually, last)
% position in the path is occupied by Goal.
%
% Of course, you would need a different definition of solution
% for a different domain.

solution([F,G,N,(X,Y)|_]) :-          % the front of the list
      mygoal(Goal),             % get the Goal
      mymap(X,Y,Goal).          % is the Goal at this X,Y?

get_closest([(X,Y,Goal)|Restof], X_Goal,Y_Goal):-
      X_Goal = X,
      Y_Goal = Y .


% generate_new_nodes(Path,Goal,Paths) finds (almost) all
% legal Paths that extend Path by exactly one step in any
% direction. A legal path is one made up entirely of
% locations that, so far as the agent can tell from the
% map, it can occupy. These are empty locations, locations
% where there are doors, locations that contain the goal
% object, or locations occupied by some object that the
% agent can remove (listed in clauses for the predicate
% can_remove/1.)

generate_new_nodes([F,G,H,(X,Y)|Rest],Paths,Goal_X,Goal_Y) :-
      findall([Fn,Gn,Hn,(XX,YY),(X,Y)|Rest],          % given these terms
      (
       Xminus is X - 1,                       % generate X,Y coordinates
       Xplus is X + 1,                        % for all the neighbouring tiles
       Yminus is Y - 1,                       % relative to the current X,Y
       Yplus is Y + 1,
       member(XX,[Xminus,X,Xplus]),           % XX becomes bound to values in
list
       member(YY,[Yminus,Y,Yplus]),           % same for YY, findall will get all
combinations
       mymap(XX,YY,Obj),                      % get whatever object is at that
location
       \+ (XX,YY) == (X,Y),                   % do not include current tile
       (
        mygoal(Obj)                           % it could be the goal (hopefully!)
       ;                                      % or
        member(Obj,[o,door])                  % it could be open space or a door
       ;                                      % or
        can_remove(Obj)                       % it can be an object he can remove
       ), get_cost(XX,YY,X,Y,Goal_X,Goal_Y,Fn,Gn,Hn,G), % G is cost to here
       \+ better_on_OPEN([F,G,H,(X,Y)|Rest],XX,YY,Fn,Gn,Hn,G)
      ),
      Paths),                                 % this is the list of possible
locations
nl,write('Paths in generate_new_nodes = ' - Paths),nl,ttyflush. % simple stub

better_on_OPEN([F,G,H,(X,Y)|Rest],XX,YY,Fn,Gn,Hn,G):-
      Rest == [] -> fail ;
        member((XX,YY),[F,G,H,(X,Y)|Rest]) ->
            (F<Fn ->
```

```
                    write('better one on open - orig - new Fn ' - X - Y - F -
Fn);
                    (write('better next node' - F - Fn), false));
      fail.

get_cost(XX,YY,Cur_X,Cur_Y,X,Y,Fn,Gn,Hn,G):-
S1 is sign(XX),                            % sign returns -1, 0 or 1
S2 is sign(X),                             % depending on sign of the term
S3 is sign(YY),
S4 is sign(Y),
(((S1=:= -1, S2=:= -1);                    % if XX & X are both negative
      (S1=:= 1, S2=:= 1);                  % or are both positive
      S1=:= 0 ;                            % or one of them is zero
      S2=:= 0) ->
      X_diff is abs(XX-X);                 % then just subtract (& abs)
((S1=:= 1, S2=:= -1);                      % if XX = pos & X = neg
      (S1=:= -1, S2=:= 1)) ->              % or XX = neg & X = pos
      (XX_abs is abs(XX),                  % then get the absolute value
      X_abs is abs(X),                     % same for X
      X_diff is abs(XX_abs + X_abs)); true),     % add them
(((S3=:= -1, S4=:= -1);                    % same thing for YY & Y
      (S3=:= 1, S4=:= 1);
      S3=:= 0;
      S4=:= 0) ->
      Y_diff is abs(YY-Y);
((S3=:= 1, S4=:= -1);
      (S3=:= -1, S4=:=  1)) ->
      (YY_abs is abs(YY),
      Y_abs is abs(Y),
      Y_diff is abs(YY_abs + Y_abs)); true),
      get_Gn(XX,YY,Cur_X,Cur_Y,G,Accumulated_Gn),      % get the real cost
      Gn is Accumulated_Gn,
      Hn is max(X_diff,Y_diff)* 10 ,       % simply take the max * 10
      Fn is Accumulated_Gn + Hn .          % f(n)=g(n)+h(n)

get_Gn(XX,YY,Cur_X,Cur_Y,Gn,Accumulated_Gn):-
      XX_abs is abs(XX),
      YY_abs is abs(YY),
      Cur_X_abs is abs(Cur_X),             % these not done yet
      Cur_Y_abs is abs(Cur_Y),
      X_diff is abs(Cur_X_abs - XX_abs),   % difference in X dir
      Y_diff is abs(Cur_Y_abs - YY_abs),   % difference in Y dir
      (ground(Gn) ->                       % if Gn is bound
      (                                    % then add to cost
      ((X_diff =:= 1, Y_diff =:= 1) ->     % if X & Y are =1 then
      Accumulated_Gn is Gn+14);            % diagonal, add 14
      Accumulated_Gn is Gn+10);            % else orthog, add 10
      Accumulated_Gn is 10) .              % else unbound, make it 1

/* % old one (simpler)
get_Gn(XX,YY,Cur_X,Cur_Y,Gn,Accumulated_Gn):-
      ground(Gn) ->                        % if Gn is bound
            Accumulated_Gn is Gn + 1 ;     % then add 1
      Accumulated_Gn is 1 .                % else unbound, make it 1
*/

% These are the objects that the agent can remove
```

```
% from a location in test7.vw.

can_remove(fruit).
can_remove(gkey).
can_remove(ykey).
can_remove(bugspray).

% mymap/3 provides an agent's map of test7.vw

mymap( -5, -19, w ).

mymap( -4, -19, w ).

mymap( -3, -19, w ).

% note that there are many more of these to describe the
% entire map in this (original ) file. I have deleted them
% here to save around 25 pages!
% look at the original code in the V-World folder.
```