V-World Workshop 4: (moving on to search)
Exercise 4: Bumble 1.3

Search Methods

Note: ***There are two relevant documents covering the vital topic of search. This one begins with Donald Nute's original introduction and leads to a discussion on what is required to extend the basic, unguided techniques, to the informed ones. The discussion is then taken further by providing some analysis of his approach and more explanation of his code. The next document "Developing best-first search" deals with how you may extend his basic search, first to greedy and ultimately to A-Star search. The explanations you may find long and overly verbose, but I have attempted to reduce ambiguity.***

Resources

search.pl	V-World's (Nute's) original depth-first & breadth-first searches
greedy_type_search.pl	Treat this as an intermediate program in the development of best-first
a_star_search.pl	Final greedy & A* searches by Graham Winstanley. Note I do not guarantee that this code is error-free, but the explanations in the accompanying document should allow you to follow by train of thought

To improve bumble's performance significantly, he needs to be able to remember where he has seen objects in his world and he needs to be able to find his way back to the resources he needs to survive. In later exercises, we will develop a version of Bumble that explores his world systematically and creates a map. But first, we will look at how Bumble can use a map to find a path to some object in his world.

Start WIN-PROLOG and open the file search.pl. At the bottom of this file, you will find clauses for the predicate mymap/3. Each clause mymap(X,Y,Object)includes an X and a Y coordinate, and the durable object (if any) located at those coordinates in test7.vw. The square that corresponds to the coordinates (0,0) marks the spot where an agent finds itself when it is loaded into test7.vw. All other coordinates are marked off from this first location of the agent.

The predicate search/2 defines a general (uninformed) search algorithm in Prolog. search/2 is designed to search the nodes in a search space until it finds a solution to the search space problem. At any given time, search/2 is passed a list of nodes that have been reached during the search as its first argument. In the first clause for search/2, the first node in the list is tested to see if it is a solution to the problem. If it is, then this node is unified with the second argument for search/2. Otherwise, the first node in the list is replaced by all the nodes below it in the search space and the search continues.

We get different kinds of search depending on how new nodes are inserted into the list of nodes yet to be processed. Blind search, whether depth-first, depth-first, or iterative deepening, does this without taking into account any information about the domain. Heuristic search methods like best-first or A* use domain information to determine the order new nodes will be explored [* note from GW: I consider A* a type of best-first search *].

We will use search methods together with our map of test7 to find paths from any location to the location of any kind of object in this virtual world.

The way new nodes are generated during search depends on the domain and determines the search space that corresponds to our problem. In this case, each node represents a path in our virtual world, and new nodes will be paths which extend a previous path by a single step. We can only add a new location to a path if that location is empty (marked on the map with o) or if it is occupied by some object that the agent can either pick up or consume (recorded in clauses for the predicate can_remove/1.) We have used a little bit of domain knowledge in defining generate_new_nodes/2 since we do not allow paths which loop back upon themselves.

For depth-first search, the new nodes are inserted at the beginning of the list of nodes to be processed. For breadth-first search, they are inserted at the end of the list. For heuristic search, we must calculate the estimated goodness of each node and use that to decide where each node goes in the list. In our Prolog code, we assert depth_first/0 or breadth_first/0 to tell insert_nodes/2 which method to use.

When we call find_path(+SearchType,+Goal,+Start,-Path), we replace SearchType with depth_first or breadth_first to tell Prolog which search method to use. Goal is replaced by tree, cross, or any other object we want to find. Start is replaced by a list of initial paths [[(X,Y)]] where (X,Y) are the coordinates of the agent's current location. If the routine is a success, Path is matched to a path from the agent's current location to the location of a goal item.

Exercises (original)

Note: although I don’t expect you do the exercises ‘as-is’, I do expect you to think about how you might go about them. When you have put some thought into the problems posed, you can read the rest of this document and then move onto the accompanying one “Developing best-first search.”

4.1 Try out the different (blind) search methods. What are your observations? Which would you use in later versions of Bumble? Why?

4.2 Think about how you would compute the actual cost of any path by adding 10 for each horizontal or lateral move and 14 for each diagonal move (the strength it would cost Bumble to follow the path.) Come up with a reasonable heuristic for computing the least additional cost it would take to extend any path to the goal location nearest to the current end of the path. Now when we generate new nodes in our search, we will include an extra element at the head of the list of moves: a number representing either the actual cost of the path (g), the estimated cost of completing the path (h), or a combination of the two (g + h). We can then use this value to sort any set of nodes (paths). Use these ideas to define new clauses for generate_new_nodes/2 and insert_nodes/2 that will implement uniform cost search, best first search, and A* search.

GW's take on all this

Nute's search algorithms are good. They are generic blind (uninformed) methods, with modifications to cater for V-World's grid & the fact that, in his approach, bumble has no way of 'finding out' where he/she is in the x-y coordinate system. This is a reasonable approach since (later) bumble will have to explore the world, and x-y coordinates chosen for visiting would be defined by him in relation to where he is now, i.e. starting position, anywhere in the world = 0,0 – so moving left (west) = -1,0

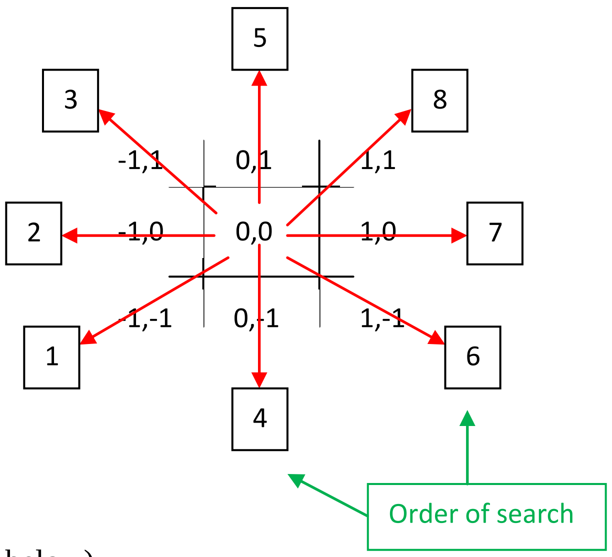
All the generic search components are here:

Test for goal: is the next tile the (a) goal. As you might expect, this is in a high-level predicate **search/2**

Generate next nodes: all those tiles (nodes in the search space) that are accessible on the next iteration. In V-World this is accomplished by the predicate **generate_new_nodes/2**. The predicate has a test for loops – it fails if bumble has visited that tile before.

From the x,y coordinates of bumble's current position, the two member/2 clauses in the 'findall' section enforce the following order (just because of the order that has Xminus, X, Xplus - & same for Yminus, etc. The order that the two member functions themselves appear are significant in this too:

X	Y
-1	-1
-1	0
-1	1
0	-1
0	1
1	-1
1	0
1	1



It is instructive to examine how this works (see below).

Decide on the next move: as each new set of nodes is generated as possible next moves, it is necessary to make a choice – in V-World's case between a possible 8 moves (horizontal, vertical & diagonal). This is achieved in the two given blind methods by the way in which the new (next possible) tiles are put onto the data structure that is used in the decision-making process. Effectively the top-most search predicate 'search/2' is the boundary condition for the search and has a test for solution, which is 'is the solution (tile) the first on the list of paths generated & concatenated during the whole process. This means that the 'insert_nodes/3' clause is really important – it makes the difference between depth-first and breadth-first search (& later best-first) behaviour.

Because the ‘insert_nodes/3’ is so crucial to which tile is placed first on the list, for informed search, it is here that you might want to do some kind of sort on the given list tiles on OPEN. I know, OPEN hasn’t been defined yet, but you can assume at this stage that the one list of tiles on the possible list is actually the OPEN list.

Some notes & explanations of the (original) code

This is the top-level goal. Invoke it with something like:

```
% find_path(breadth_first,tree,(0,0),_).
```

It does some initial housekeeping, followed by an assertion of the goal we are trying to achieve, and then it calls the top-level search predicate. Note that because of the way search proceeds, it always puts the frontier of the search (the ‘chosen’ next node), for a path from finish to start, the path needs to be reversed. It seems a reasonable thing to do.

```
find_path(SearchType,Goal,Start,Path) :-
      retractall(search_type(_)),          % house keeping
      assert(search_type(SearchType)),
      retractall(mygoal(_)),
      assert(mygoal(Goal)),                % assert the given goal
      search([[Start]],ReversedPath),      % call the top-level search
      reverse(ReversedPath,Path).          % reverse it start-to-goal path
```

This is the top-level search predicate & is actually the boundary condition for search. search/2 has the test for solution, which will terminate the process

```
% call with (e.g.)
%         find_path(breadth_first,tree,(0,0),_).

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

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

search([FirstNode|RestOfNodes],Solution) :-     % from the current tile
      generate_new_nodes(FirstNode,NewNodes),   % generate next poss tiles
      insert_nodes(NewNodes,RestOfNodes,NextSetOfNodes),     % add to OPEN
      !,

      search(NextSetOfNodes,Solution).          % recurse with new list
```

This is the part that generates all possible legal next moves. It uses the amazing ‘findall/3’ predicate that returns an unsorted list of all the solutions possible for a given goal. The first argument in findall/3 (in generate_new_nodes/2) is a list of terms, the second contains the actual goal – the conditions for choosing next moves, and the 3rd is the list of solutions. findall/3 could be implemented in other ways, with much backtracking, but it simply does it all for you – it seems to have been made for this!

```
generate_new_nodes([(X,Y)|Rest],Paths) :-
      findall([(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]),          % findall will get all combinations
       mymap(XX,YY,Obj),                     % get the object at that location
       \+ member((XX,YY),[(X,Y)|Rest]),      % been here before? (loop detection)
       (
        mygoal(Obj)                          % it could be the goal (hopefully!)
       ;                                     % or
        member(Obj,[o,door])                 % could be open space or a door
       ;                                     % or
        can_remove(Obj)                      % it can be an object he can remove
       )
      ),
      Paths).                                % list of possible locations

% 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).
```

Depth-first or breadth-first. Remember that it is here that the front of the list is determined for the solution test in search/2, i.e. you can add more sophistication here for informed 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
```

The map

The (very many) assertions that collectively make up the complete map of the world test7.vw has been created so that we can concentrate on the actual search problem, rather than having to worry about how the map could be represented (or eventually 'discovered'). It is, of course, unreasonable to expect us to create this for each & every world we create, but it could easily be automated. Each tile (cell, location) in any V-World world has an X-Y coordinate (at its centre), so we could use that in some kind of scanning routine & complete the map, or as we will see later, we could rely on bumble to build it up as he explores. In this simple version of search, the agent default starting location is set to 0,0 and every other tile is annotated relative to it. So for example, the tile to bumble's immediate right (east) is 1,0

```
% mymap/3 provides an agent's map of test7.vw
% this comprises a complete map of test7.vw
% it would be expected that later versions of
% bumble could create this as it explores

mymap( -5, -19, w ).

mymap( -4, -19, w ).

mymap( -3, -19, w ).

% and so on for the complete map
```

One last set of comments on this search program

This is a clever piece of programming in Prolog. It is simple but effective and pretty optimal. It could be improved with world boundary checks, but since bumble could never consider a tile which is beyond the boundary, due to the constraints on the world, i.e. that they are all completely bounded by walls or doors, the problem does not arise.

It is instructive to see the paths generated by the two blind search methods. Try for example:

```
?- find_path(depth_first,tree,(0,0),Path).
```

In test7.vw, the goal (the tree) is right next to bumble on his left, so search should find it immediately (you would imagine!). However, if you use depth-first as above, you will notice that bumble bumbles along right past it & takes a tour of that section of the world before coming back to the tree. It gets there eventually, but initially it would seem to be doing a really bad job of finding the tree. You need to think in terms of depth-first, i.e. going deep into the chosen branch. The first branch that bumble is forced to consider is south-west (look at the search order in my table and diagram above). Once this apparently erroneous decision has been taken, poor bumble is destined to go right past his goal. He will act systematically, but hardly in any way optimally. However, if you run the breadth-first version, bumble finds it in a much more convincing way (just for this scenario of course – with the goal very close to the root).

We need now to consider best-first search. Please refer to the accompanying document "Developing best-first search."

Exercises based on the program 'a_star_search.pl'

After you have read the document "developing best-first search", try out the following exercises:

1. Open & compile the program 'a_star_search.p' and examine the code, noting the function of each predicate. Put sypoints on whichever predicates you would like to trace. Use the query (say): `find_path(a_star,door,(0,0),_).` Some examples are
    a. Generate_new_nodes/4. Note especially how new tiles are generated in the findall/3 clause. Make sure to use Leap or Skip when you get to the goal `mymap(XX,YY,Obj)`, otherwise you'll be tracing all day.
    b. get_cost/10 and get_Gn/6
    c. insert_nodes/3
    d. and of course search/6
2. Try altering the map to make it more challenging
3. Try using Nute's 'how_long/3' predicate. An example would be
    a. `how_long(find_path(a_star,door,(0,0),Path),Duration,Error).`