*An approach to best first… eventually to full A**

This document accompanies Workshop 4 and should be read after reading and understanding that.

This document discusses the requirements for best-first search: greedy search using only h(n), and A* which uses h(n) and g(n) [refer to lecture notes on search]. It basically takes you on a 'journey' through my development of Nute's original systematic searches (so correctness and/ or optimality not guaranteed).

The programs can be used in your development of V-World in workshop exercises and the assignment (with acknowledgements).

Resources:	greedy_type_search.pl	up to the development of full A*
	a_star_search.pl	covers the development of full A*

There are changes that must be made to the data structures to accommodate the following:

h(node) = a heuristic estimate of how close the node is to a goal (this is used for greedy)

g(node) = the actual cost (so far) of getting from start to that node (will not use this yet)

Starting with the top-most predicate 'find_path'

```
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([[Cost,Start]],ReversedPath,X_Goal,Y_Goal,[],[]),
       reverse(ReversedPath,Path),                    % reverse it - start-to-goal
       nl,write('Path = ' - Path),nl,ttyflush, ! .    % print it - final path
```

The first change is the 'findall/3' goal. It is used here to get all occurrences of the goal that it can find in the current map. Remember that bumble will eventually be responsible for generating this map & will have recorded where everything that he finds is. This is followed by a clause that returns the closest goal to where bumble presently is. Care is needed here though – see the comments in the a_star.pl code. search/6 is then called. Notice that we now have CLOSED and OPEN

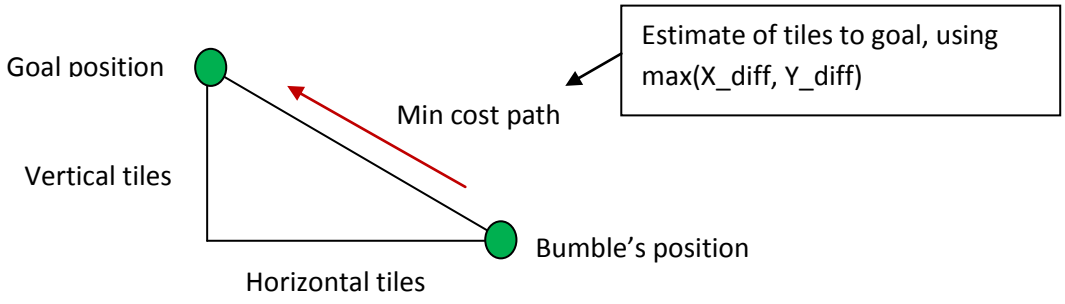
```
:- dynamic [search_type/1, mygoal/1].

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

search([FirstNode|RestOfNodes],Solution,Goal_X,Goal_Y,CLOSED,OPEN) :-
       generate_new_nodes(FirstNode,NewNodes,Goal_X,Goal_Y),         % next poss
       insert_nodes(NewNodes,RestOfNodes,NextSetOfNodes),     % add to OPEN
       !,
       add_to_CLOSED(NextSetOfNodes,CLOSED,New_CLOSED),       % add to CLOSED
       search(NextSetOfNodes,Solution,Goal_X,Goal_Y,New_CLOSED, NextSetOfNodes).


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

As before, the FirstNode is the frontier of search. search/6 is sent the X & Y coordinates of the chosen goal (the closest – Goal_X & Goal_Y) and the CLOSED and OPEN lists. The first occurrence of search/6 is the boundary condition for recursion & really only checks if the FirstNode is the goal

New possible nodes (tiles) in the search need to be generated… as below:

```
generate_new_nodes([C,(X,Y)|Rest],Paths,Goal_X,Goal_Y) :-
       findall([Cost,(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 whatever object is at that location
        \+ member((XX,YY),[(X,Y)|Rest]), % have we been here before? (loop detect)
        (
         mygoal(Obj)                       % it could be the goal (hopefully!)
        ;                                        % or
         member(Obj,[o,door])                    % it could be a door or open
        ;                                        % or
         can_remove(Obj)                         % it can be an object he can remove
        ), get_cost(XX,YY,Goal_X,Goal_Y,Cost)    % h(n) calculation
       ),
       Paths).                             % this is the list of possible locations
```

This is very close to the original. Notice that a Cost variable is now used, and it is used within findall/3 to assign a heuristic estimate (called Cost here for simplicity) of how close that tile is to the chosen goal. In get_cost/5, the Cost variable is calculated using a very simple heuristic, i.e. take the maximum difference between the number of tiles in the X direction and the number of tiles in the Y direction.



There is one complication however. Tile positions are labelled relative to bumble's starting position, which is (0,0), so we have positive and negative numbers to deal with. The predicate 'get_cost/5' below takes this into account – see comments in the code:

```
get_cost(XX,YY,X,Y,Cost):-
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 in this case
(((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),
      Cost is max(X_diff,Y_diff).        % simply take the max
```

For A* this is good since it will never overestimate the cost of getting to the goal, it is admissible (need that for A*). However, there will be paths with an identical cost, and in these cases the paths will be ordered according to the inherent tile choice order in 'generate_new_nodes/4'. It would be possible (& useful) to be able to move away from the uniform cost approach used here & in fact Nute suggests applying different costs to diagonal and vertical/horizontal moves. We could also imaging applying costs to each tile according to some predefined algorithm, e.g. higher costs for tiles adjacent to hornets (say).

At the end of generate_new_nodes/4, Paths, for a goal of 'gkey' at (3,-2) and a wall at (1,-1) will have the format:

```
[[4,(-1,-1),(0,0)],
[4,(-1,1),(0,0)],
[3,(0,-1),(0,0)],
[3,(0,1),(0,0)],
[2,(1,0),(0,0)],
[3,(1,1),(0,0)]]
```

All possible moves for bumble from (0,0), along with the heuristic estimate at head of each path (list). You can also see here why the path should eventually be reversed.

If you take a look at the world we are dealing with (see graphic later in this document), you will see that there is a tree at position (-1,0) and a wall obstacle at position (1,-1). Therefore there are only 6 next possible moves for bumble from (0,0) – not the maximum 8.

i.e. each tile (here from a starting position of (0,0) is recorded, along with the heuristic estimate calculated in get_cost/5.

Next in search/6 is the 'insert_nodes/3' clause. For greedy search, this is defined very simply as a sort:

```
% 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 [cost,[X,Y, X,Y...]] and uses the first element
% of each path [1] as the sort key. Ascending order too

insert_nodes(Set1,Set2,Sorted):-
      search_type(greedy),     % if it is greedy, do sort
      append(Set2,Set1,Set3),
      sort(Set3,Sorted,[1]).   % do something clever!
```

sort/3 sorts the items in Set3 in ascending order, using the 1st element of each item (i.e. the cost… f(n)) as the sort term. It puts the resulting sorted list into 'Sorted'.

Next in search/6 is the clause 'add_to_CLOSED/3'. This is required to add the tile just committed to to CLOSED, and remove it from the list 'NextSetOfNodes' (which becomes bound to OPEN in recursion). It sends the 'NextSetOfNodes', the current 'CLOSED' list and a placeholder for the updated CLOSED, called here 'New_CLOSED'.

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

Several important list handling operations happen here. The sorted list 'NextSetOfNodes' has been sent as the first argument & in add_to_CLOSED/3, by specifying this variable as [First|Rest] we are taking apart that list right away into the head and the tail. The head is then appended to the CLOSED list, which is now called 'New_CLOSED' and this provides the binding required for search/5. It is made into a list in the append/3 clause:

```
      append([First],CLOSED,New_CLOSED),
```


so that CLOSED contains a list of paths (a list of lists effectively).

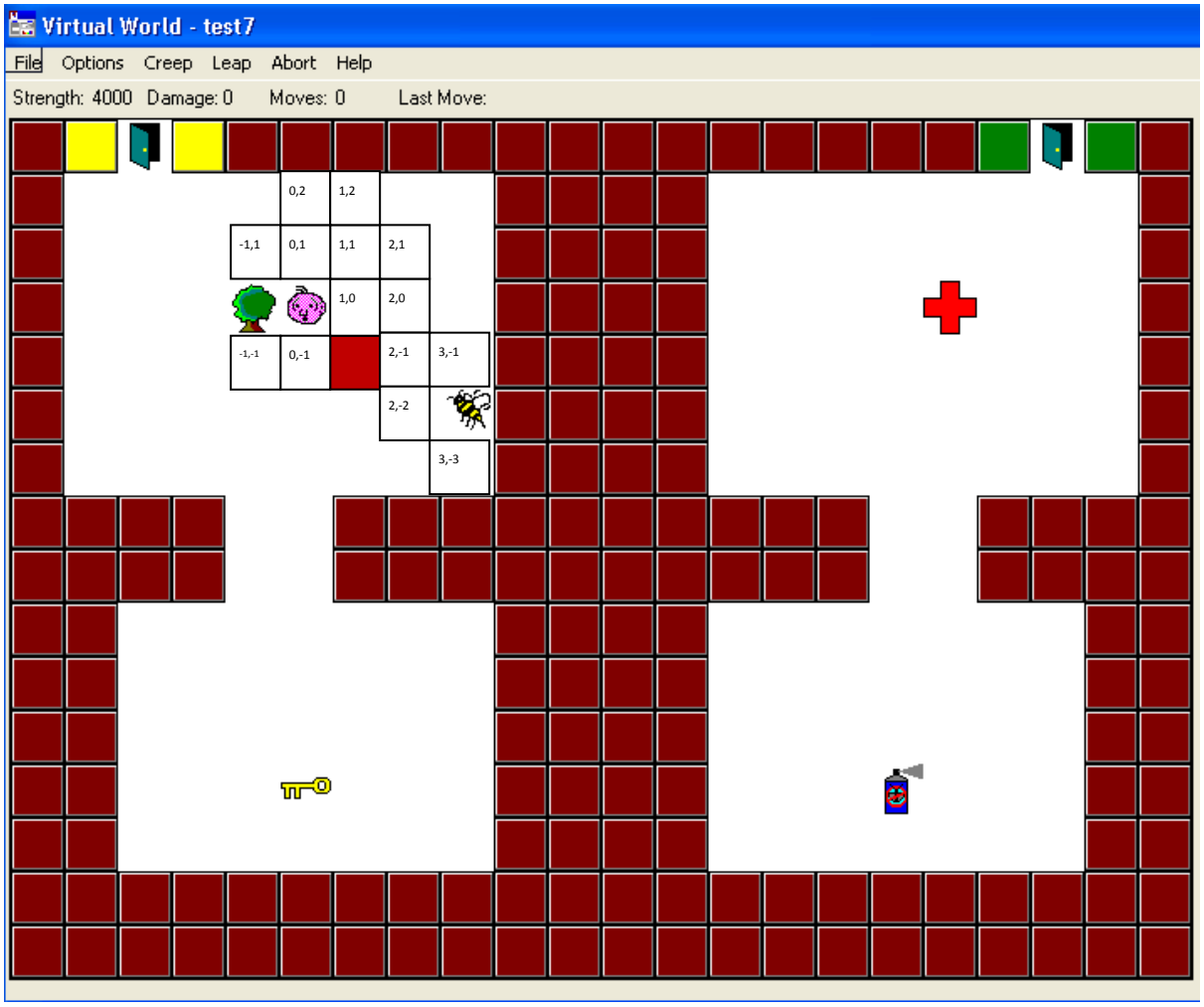
'search/6' now recurses with the new 'NextSetOfNodes', i.e. becomes OPEN.

Examine the operation of the latest search

Bumble is at (0,0), i.e. the starting position. Every tile is relative to whatever starting position he is in. There is a tree at (-1,0), a hornet at at (3,-2), a door at (-3,3) and a wall obstacle at (0,-1).

We are going to find a path from where bumble is to the hornet:

```
?- find_path(greedy,hornet,(0,0),_).
```

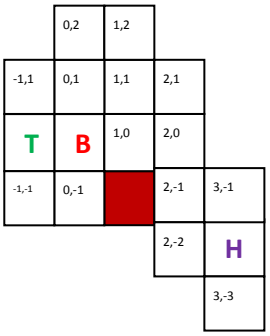


You can make up your own (complete) version of the grid if you like. It is instructive to be able to follow bumble as the search proceeds. I have shown the OPEN & CLOSED lists below, as well as the final path. In my development of this program I have made much use of 'write/1' and message boxes to show the progress of all the data structures. They have been taken out for the final version of course.

```
OPEN        [[2,(1,0),(0,0)],
             [3,(0,-1),(0,0)],
             [3,(0,1),(0,0)],
             [3,(1,1),(0,0)],
             [4,(-1,-1),(0,0)],
             [4,(-1,1),(0,0)]]

CLOSED:     [[2,(1,0),(0,0)]|_43655]

OPEN:       [[1,(2,-1),(1,0),(0,0)],
             [2,(2,0),(1,0),(0,0)],
             [3,(0,-1),(0,0)],
             [3,(0,1),(0,0)],
             [3,(1,1),(0,0)],
             [3,(0,-1),(1,0),(0,0)],
             [3,(0,1),(1,0),(0,0)],
             [3,(1,1),(1,0),(0,0)],
             [3,(2,1),(1,0),(0,0)],
             [4,(-1,-1),(0,0)],
             [4,(-1,1),(0,0)]]
```

B=Bumble, T=tree

H=hornet




```
CLOSED:     [[1,(2,-1),(1,0),(0,0)],
             [2,(1,0),(0,0)]|_43655]

OPEN:       [[0,(3,-2),(2,-1),(1,0),(0,0)],% this is the goal. Cost = 0
             [1,(2,-2),(2,-1),(1,0),(0,0)],
             [1,(3,-1),(2,-1),(1,0),(0,0)],
             [2,(2,0),(1,0),(0,0)],
             [2,(1,-2),(2,-1),(1,0),(0,0)],
             [2,(2,0),(2,-1),(1,0),(0,0)],
             [2,(3,0),(2,-1),(1,0),(0,0)],
             [3,(0,-1),(0,0)],
             [3,(0,1),(0,0)],
             [3,(1,1),(0,0)],
             [3,(0,-1),(1,0),(0,0)],
             [3,(0,1),(1,0),(0,0)],
             [3,(1,1),(1,0),(0,0)],
             [3,(2,1),(1,0),(0,0)],
             [4,(-1,-1),(0,0)],
             [4,(-1,1),(0,0)]]

CLOSED:     [[0,(3,-2),(2,-1),(1,0),(0,0)],
             [1,(2,-1),(1,0),(0,0)],
             [2,(1,0),(0,0)]|_43655]
```

You could try it for the tree and the door.

This is not really a full greedy search. It assumes what Yoav Shoham calls "everywhere admissibility":

"if, whenever it extends a path from a node X to its neighbours, it has already found an optimal path to X." [ref: Shoham, Y, (1994). AI techniques in Prolog. Morgan Kaufmann.].

This means that, whenever a node in the search tree is chosen, it is committed to – no changes will be made to it. It also means that there would be no (need for a) check in CLOSED to see if a better, less costly, path had already been seen to the node in question. Shoham states that situations commonly arise when this simple approach is appropriate. We will extend the present program to cater for full greedy & eventually A*. This will involve the necessary check, at each iteration, for a better path on OPEN and CLOSED.

Extending the program to check for possibly better paths already considered (A will be developed from this – see best_first_search.pl)*

We have the necessary means to do this: OPEN and CLOSED, plus data structures that contain paths as lists with the path cost at their heads:

1. We have already calculated and assigned h(n) to each possible move as tiles estimated from the current node (tile) to the goal.
2. We have already made a choice of next tile in the search as the smallest h(n). We did this by a simple sort on the 'insert_nodes/3' predicate for 'greedy'.
3. We have already created a list which progressively contains those paths already visited called CLOSED and there is an OPEN list that contains the paths, with a next node, that are yet to be considered.

The following is a list of the steps in sequence that we need to do to make sure our search engine always takes the next best node (path).

1. After 'generate_new_nodes/4', and 'insert_nodes/3', check that any nodes generated that are already on the OPEN list have a better (smaller) cost than the current one being considered. Since we do a simple sort on the smallest h(n), this will not be implemented (yet). Remember that there is a loop detection heuristic still built into 'generate_new_nodes/4'.
2. For each move, check that a 'better' path exists on CLOSED to the next node chosen in our simple sort. We do need to do this and if we find one, it should be moved back to OPEN & it should then be sorted again (not strictly necessary since 'append/3' will automatically place this (better) path at the top, i.e. the next path to be extended). The path that this is replacing should be placed on CLOSED. There is no need to sort this list.

```
% There may be multiple paths to the same node - Use findall/3
/*
This is a simple test for the predicate, just to make sure it does
what it's supposed to. It is quite general.

check_CLOSED([[2,(1,0),(0,0)], [3, (2,2),(1,1)]],
      [[4,(1,1),(0,0)], [3,(3,4),(4,2)],
      [1.9,(1,0),(0,0)],[6,(6,0),(6,8)], [1.8,(1,0),(0,0)]], N).

there are 2 occurrences of [_,(1,0),(0,0)] in CLOSED, one with Cost=1.9 &
the other 1.8
*/

check_CLOSED([First|Rest],[D|CLOSED],NextSetOfNodes_2):-     % strip off the
                                                             first (best?)
      CLOSED = [] -> NextSetOfNodes_2 = [First|Rest]; % empty? Do nothing
      (First=[A,B|Closed],                  % gets the cost & head of First
      findall(Best_path,
      (
      member([C1,B|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 = [C|_],                         % retrieve the cost of best
      C<A ->                                % is the cost smaller on CLOSED?
      append([Best],Rest,NextSetOfNodes_2);  % append as a list.. [First]
      NextSetOfNodes_2 = [First|Rest].      % else send original list back
```

This clever little predicate deserves some explanation. Too often I come across Prolog code that uses sophisticated list processing, and it is assumed the reader can follow it in the same way as the programmer. In fact there are always alternative ways to do things, and I have tried here to do it as simply (& logically) as possible.

1. call check_CLOSED/3 with the OPEN and CLOSED list. The first argument in check_CLOSED/3 strips off the first node in OPEN. We need that since this is the next node to be expanded – after sort. The second argument strips off the head of CLOSED. We are actually interested in the tail of CLOSED because the next node to be expanded has already been added to CLOSED. The third argument will be the new OPEN list (it may of course be no different to the original if no better nodes are found on CLOSED.

2. if CLOSED is empty, then do nothing – just send the original list back

3. `First=[A,B|Closed],` gets the two first elements of First, which are the cost and the top node of the first element in the OPEN list prvided

   a. A = 2 (say)
      B = (1,0), say

4. the findall/3 clause is perhaps overkill, but I am not totally convinced that there would only be one path to the current chosen node on CLOSED. findall/3 will

find all the paths on CLOSED that match the top node (the current chosen tile). The tests are:

```
member([C1,B|Closed],CLOSED,Pos),  % is the node (B) in CLOSED?
mem(CLOSED,[Pos],Best_path)        % gets the path at Pos
```

The first clause here checks if the node (1,0), say, exists on CLOSED. The Pos Variable will be bound to the position of this in the CLOSED list.

The second clause uses the list position number to retrieve the path and binds that to 'Best_path'

findall/3 will keep doing this until there are no more occurrences. If it turns out there is only one – fine – it will find one.

5. At this point we may have one or more elements (paths) in Paths. We will use the same sort/3 predicate as used in insert_nodes/3. It sorts on the first element of each path list, which is the path h(n) of the current node.

```
sort(Paths, Sorted_paths,[1]
```

6. Now get the head of the sorted list, which is, of course, the best, and the head of that list is the path cost.

```
Sorted_paths = [Best|_],              % retrieve the best path
Best = [C|_],                         % retrieve the h(n) of best
```

7. Now the test: is the cost of the best path on CLOSED smaller (better) than the current node: If it is, then append it to the rest on OPEN (as a list). This becomes the head of OPEN, i.e. it replaces the current one – with the current one now safely on CLOSED. If it is not better, then put the original list together again & send it back as it was when the predicate was called – just like we did for the empty CLOSED list.

```
C<A ->                                % is the cost smaller on CLOSED?
append([Best],Rest,NextSetOfNodes_2);  % append as a list.. [First]
NextSetOfNodes_2 = [First|Rest].
```

Extending to full (general) A-Star

There are a number of changes that must now be made to the greedy_search we have up to this point (see earlier notes about the current search's limitations.

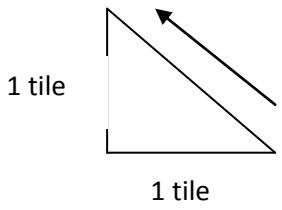
1. The data structures that represent the developing paths need to be changed. Up to this point a single Cost variable has been used, and this has been calculated as h(n). Now we need to take account of the cost so far on the developing path g(n) and for A*, the evaluation function f(n) = h(n)+g(n). We could simply change our Cost variable to be the final f(n) value and sort the list of paths on that, but since this is an educational program, I have added Gn and Fn to the head of the path list data structure. Below is the top-level call to search/6. Of course each predicate & calling goal that uses this structure will need appropriate extensions (tedious but simple).

```
search([[Fn,Gn,Hn,Start]],ReversedPath,X_Goal,Y_Goal,[],[]),
```

2. A check needs to be made for potentially better paths on OPEN in just the same way that a check was made on CLOSED. We need to do this though when new nodes are generated in 'generate_new_nodes/4'. The test should be quite simple – if a better path exists on OPEN, don't generate a worse (new) one. We will replace the simple heuristic: '`\+ member((XX,YY),[(X,Y)|Rest])`' with a more appropriate one for A*.

3. Since g(n) is the accumulated 'real' cost of getting to the current node using the related path, we no longer need (or want) the loop detection clause in generate_new_nodes/4, i.e. if we have been there before, we shouldn't just forbid it, we should allow the system we have just set up in (2) above to check if it was a better path.

4. We need to put in place a means for g(n) to be calculated in an additive (accumulated) way, and for f(n) to take this into account as well as h(n)… f(n)=g(n)+h(n). We ideally want to be able to add some sophistication to the value of g(n), so the predicate that makes the calculation should be equipped with sufficient information for this. For example, Nute suggests that g(n) for each move should be 10 for any horizontal or vertical move and 14 for any diagonal move. This is simple Pythagoras:



Diagonal distance = √(1+1) = 1.414

≅ 1.4 … X 10 =14

… Still better than 1+1 tile moves = 20

For this the g(n) calculating predicate needs to know the X,Y coordinates of the current tile and the X,Y coordinates of the suggested move.

The call to the get_cost predicate in generate_new_nodes/4 has been extended to cater for `Fn,Gn and Hn`, and the variable G at the end of the argument list is bound (in generate_new_nodes/4) to the value of g(n) up to this point.

```
generate_new_nodes([F,G,H,(X,Y)|Rest],Paths,Goal_X,Goal_Y) :-
……
……

), get_cost(XX,YY,X,Y,Goal_X,Goal_Y,Fn,Gn,Hn,G) % G is the cost to here
      ),
      Paths).
```

The new get_cost/10 predicate now includes methods for getting values for h(n), g(n) and f(n). h(n) is calculated in exactly the same way as before – using the same heuristic, and f(n) is very simply defined as the addition of g(n) & h(n) – as you would imagine. g(n) however needs to be cumulative, and it needs to be extendable to get costs in ways described in (4) above. Therefore a new predicate has been defined – get_Gn/6 so that this can be accommodated:

```
get_cost(XX,YY,Cur_X,Cur_Y,X,Y,Fn,Gn,Hn,G):- % XX & YY are poss moves, X&Y
are goal positions and X_Cur & Y_Cur are where he is currently
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),                    % simply take the max
Fn is Accumulated_Gn + Hn .                  % f(n)=g(n)+h(n)
```

New from here

We could use a very simple algorithm that merely sets g(n) to 10 (say) initially (when Gn is unbound), and then add 10 to it each time a tile is added to the developing path. Below is just about the simplest I can think of:

```
get_Gn(XX,YY,Cur_X,Cur_Y,Gn,Accumulated_Gn):-
      ground(Gn) ->                          % if Gn is bound
            Accumulated_Gn is Gn + 10 ;      % then add 10
      Accumulated_Gn is 10 .                 % else unbound, make it 10
```

To incorporate the different costs associated with orthogonal (vertical or horizontal) and diagonal moves, a slightly more involved predicate is needed:

```
get_Gn(XX,YY,Cur_X,Cur_Y,Gn,Accumulated_Gn):- % Gn is cost so far
      XX_abs is abs(XX),
      YY_abs is abs(YY),
      Cur_X_abs is abs(Cur_X),              % make them absolute
      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
```

In this version the absolute values of the XX, YY, X & Y values is calculated & a simple check is made for a diagonal move, i.e. if both the X & Y difference is 1, then it must be diagonal, so therefore add 14 to the cost so far and bind it to the variable `Accumulated_Gn`. Otherwise it must be orthogonal so add 10 instead. The test for ground is still there from the simplest version & again if this fails, the `Accumulated_Gn` is just set to 10.

Testing A-Star

This can be done in a number of ways. During the development of this search program, testing was carried out by using:

1. Spypoints on new or crucial predicates. Judicious use of 'creep', 'skip' and 'leap' make this process less 'nightmarish' than it could be.

2. Stubs. These are message boxes and writes. Care needs to be taken with message boxes though. If a loop or really long sequence occurs, there is no way to break it (ctrl Break)

3. Test scenarios for new predicates (see check_CLOSED/3 for an example of this). This method is especially useful for complex list processing predicates & where it is important to be able to use Trace on a'sanitised' experimental set of input lists, etc.

The final tests were carried out in V-World itself, using my modifications to Nute's test_7. It is possible to place any object anywhere on the map using his long list of clauses that make

up the map that the search needs. See the map visualisation of this earlier. Care here**, it is not exactly the same as test_7**

```
| ?- find_path(a_star,door,(0,0),Path).

Final Path =  - [(0,0),(-1,1),(-2,2),(-3,3),0,38,38]
Path = {0,0, -1,1, -2,2, -3,3, 0, 38, 38}

| ?-
```

And:

```
| ?- find_path(a_star,hornet,(0,0),Path).

Final Path =  - [(0,0),(1,0),(2,-1),(3,-2),0,38,38]
Path = {0,0, 1,0, 2,-1, 3,-2, 0, 38, 38}

| ?-
```

As an exercise, try setting up challenging search scenarios for bumble.

One other useful resource in Nute's original code is the predicate 'how_long/3'. It returns the time taken to complete a goal in milliseconds:

```
| ?- how_long(find_path(a_star,hornet,(0,0),Path),Duration,Error).

Final Path =  - [(0,0),(1,0),(2,-1),(3,-2),0,38,38]
Path = {0,0, 1,0, 2,-1, 3,-2, 0, 38, 38} ,
Duration = 149 ,
Error = 0
```

So we could use this for any new facility we want to give Bumble.

Finally – have fun with these search routines! Feedback always appreciated.

End of this document (for now)!