CAP 5602
Summer, 2011

Lesson 9: Heuristic Search and A* Search

The topics

1. Heuristic Search
2. The A* Search
3. An example of the use of A* search.

1. Heuristic Search

The idea of heuristics is to attach to each state a number that indicates how far that state is from a goal state.

For example, for the 15-puzzle two heuristics are the number of tiles that are out of place and the sum of the Manhattan distances, i.e. the number of swaps that would take a tile from its present position to the final one.

Let us look at the graph from Figure 1. The number next to the state is its heuristic value, an estimate of how long it would take to get from the current state to a goal state. For the goal nodes the heuristic value is 0.

A greedy traversal would take into account only the heuristic value of the state. It has a list of live nodes and a list of dead nodes. The algorithm works as follows:

1. At the beginning only the start node is active and the list of dead nodes is empty.

2. If the list of active nodes is empty, stop with failure.

3. We select a node with the smallest heuristic value from the list of live nodes. This is the **expansion** node. If the expansion node is a goal, stop with success.

4. We generate all children of the expansion node. The ones that are not dead are added to the list of live nodes.

5. The expansion node is added to the list of dead nodes and we go back to step 2.

Let us see how this algorithm works on the graph from Figure 1 when the start node is a. The number next to the arc is the cost of the transition and the one that follows the node is its heuristic value. The goal state is g.

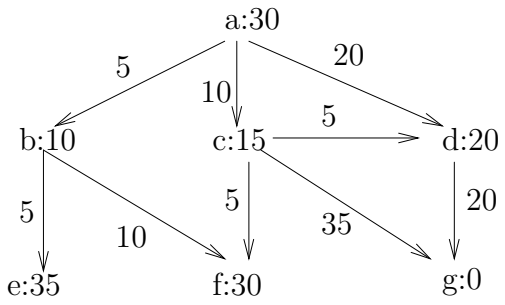

Figure 1: The greedy method

We start with a and expand it by finding its children, b:10, c:15, d:20. This is the list of **active nodes**. We move a to the dead list and continue.

We select the child with the smallest heuristic value from the active nodes and expand it. The children of b are e:35 and f:30. We remove b from the active list and add the new nodes to the live nodes.

Now, the active nodes are c:15, d:20, e:35, and f:30. The node c has the smallest value, so it becomes the expansion node. Its children are f:30, d:20, and g:0. The nodes f and d are already active, so we add only g. Then we move c to the dead list.

We continue. The actice list is d:20, e:35, f:30, g:0. The node g gas the smallest value, so it is the expansion node. Since g is a goal node we stop.

We notice two things. If we want to find the path from the start node to a goal, we need to store the paths from the start node in the active list, not just the node.

The second observation is that the greedy method does not get a path of minimal cost. It will get the path [a,c,g] of cost 45, when the is a cheaper path, [a,c,d,g] that costs only 35.

2. The A* algorithm

```
% bestFirst(Start, Solution): Solution is path from Start to a goal
bestFirst(Start, Solution):- % assume 9999 is > any f-value
    expand([], l(Start, 0/0), 9999, _, yes, Solution).
% expand(Path , Tree, Bound, Tree1, Solved, Solution):
% Path is a path from the start node to the subtree Tree
% Bound is the f_limit for expansion of Tree
```

```
% Tree1 is Tree expanded within Bound; the f-value of Tree1 is
% greater than Bound (unless a goal node has been found during the
% expansion)
% Solved has values yes, no, or never
% Solution is path from the start node 'through Tree1' to a goal
% node within Bound
% Path, Tree, Bound are the input parameters of expand, i.e. they are
% instantiated at the time of the call.
% expand produces 3 types of results depending on the value of Solved
% If Solved = yes, then Solution is a solution path and Tree1
% is uninstantiated.
% If Solved = no, Tree1 = Tree expanded, so its f-value exceeds Bound
% Solution is uninstantiated.
% If Solved = never, Tree1 and Solution are uninstantiated.
% if goal is found, Solution is solution path and Solved = yes
% Case 1: goal leaf node, construct a solution path
expand(P, l(N, _), _, _, yes, [N|P]) :- goalReached(N).
% Case 2: leaf-node, f-value less than Bound
% Generate successors and expand them within Bound
expand(P, l(N, F/G), Bound, Tree1, Solved, Sol):-
    F =< Bound,
    (bagof(M/C, (successor(N,M,C),
    % M is a successor of N, and C is the cost of the arc from N to M
    not(member(M,P))), Succ), !,
    % put all successors of N that are not on the path P in Succ
    succList(G,Succ,Ts), % make the subtrees Ts
    bestf(Ts,F1), % f-value of best successor
    expand(P,t(N,F1/G,Ts),Bound,Tree1,Solved,Sol)
    ;
    Solved = never % the F is bigger than the bound
    ).

% Case 3: non-leaf, f-value less than Bound
% Expand the most promissing subtree; depending on
% results, the procedure continue will decide how to proceed
expand(P, t(N,F/G,[T|Ts]),Bound, Tree1, Solved, Sol) :-
    F =< Bound,
    bestf(Ts,BF), min(Bound,BF,Bound1), % Bound1 = min( Bound,BF)
```

```
        expand([N | P], T , Bound1, T1, Solved1, Sol),
        % T is the child of N with the lowest F and its F-value
        % is less than Bound1 (Bound1 is the lowest of the
        % F-values of the other children and Bound)
        % the following comments try to elucidate the continue statement
        % that follow.
        % the expand call end when Solved1 = no, yes, or never
        % if Solved1 = yes, Sol is defined and Solved is uninstantiated, so
        % the first clause of continue will instantiate Solved to yes,
        % and continue will be satisfied.
        % if Solved1 = no, T cannot be expanded within Bound1.
        % Then T1 is set to T, and the call will be unified with the second
        % clause of continue. It will reorder the subtrees of N and look
        % another subtree to expand.
        % if Solved1 = never, T is a leaf.
        % the continue statement will be unified with the third clause
        % of continue, that will simply delete these child of N an continue
        % searching the other ones.
        continue(P,t(N,F/G, [T1|Ts]), Bound, Tree1, Solved1,Solved, Sol).

    % Case 4: non-leaf with empty subtrees
    % This is dead end which will never ce solved
    expand(_, t(_,_,[]), _, _,never,_):-!.

    % Case 5: value greater than Bound
    % Tree may not grow
    expand(_, Tree, Bound, Tree, no, _):- f(Tree,F), F ¿ Bound.

    % continue(Path, Tree, Bound, NewTree, SubtreeSolved, TreeSolved, So-
lution)
    continue(_, _, _, _,yes, yes, Sol).

    continue(P, t(N, F/G, [T1 | Ts]), Bound, Tree1, no, Solved, Sol):-
        insert(T1,Ts, NTs), % the expansion node is not in T1
        bestf(NTs,F1),
        expand(P,t(N,F1/G,NTs), Bound,Tree1,Solved,Sol).

    continue(P,t(N,F/G,[_|Ts]),Bound, Tree1, never, Solved, Sol):-
```

```
    bestf(Ts,F1),
    expand(P, t(N,F1/G, Ts),Bound,Tree1, Solved, Sol).

% succList(G0, [Node1/Cost1,...],[l(BestNode, BestF/G),...):-
% make list of search leaves ordered by their f-values
succList(_,[],[]).
succList(G0,[N/C|NCs],Ts):- % G0 is the path cost of the parent of N
    G is G0 + C, % the cost of the path to N
    h(N,H), % find, H, the heuristic value of N
    F is G + H,
    succList(G0,NCs,Ts1), % find the list of the other successors,
0.2in % ordered by F
    insert(l(N,F/G),Ts1,Ts). % insert the leaf in Ts1,
    % keeping the < order on F

% Insert T into list of trees Ts preserving the order of the F-values
insert(T, Ts, [T|Ts]):- f(T,F), % get the F value of the tree T
    bestf(Ts,F1), % get the best F of the rest of Ts
    F=< F1, !. % insert at the head

insert(T, [T1|Ts],[T1|Ts1]):- insert(T,Ts, Ts1). % insert in the body

% Extract f-value
f(l(_,F/_),F). % the F-value of a leaf
f(t(_,F/_,_),F). % the F-value of a tree

% bestf is the smallest F of an ordered list of trees.
bestf([T|_],F):- f(T,F). % the list is ordered in the decreasing value of F
bestf([],9999). % there are no trees.

% min(X,Y,Z) is satisfied if Z is the smallest of X and Y
min(X,Y,X) :- X ¡ Y.
min(X,Y,Y).
```

3. How does A* work?

Let us see how A* works on the graph from Figure /ref1. We enter the query

```
?- bestFirst(a,Sol).
```

We will concentrate on the expand predicate, that generates the subtrees. The first call is

(1) expand([],l(a,0/0),9999,_192,yes,Sol).

The structure l(Node,F/G) means that Node is a leaf, F is its heuristic value plus the cost of the path from the start node to Node, and G is the cost of the path from the start node to Node. The call means that we can expand the tree with root a to an F $\leq$ 9999. We can enlarge this number if it is not sufficient.

_192 is a system variable.

This goal fails the first clause of the expand because the node a is not a goal. We match the goal with the second clause of expand. This forms a list of the successors of a ordered in the increasing order of their F values. The F value of a is the smallest of the F-values of its children. So, now we have the goal

(2) expand([],t(a,15/0,[l(b,15/5),l(c,25/10),l(d,40/20)]),9999,_192,yes,Sol).

t(a,15/0,[l(b,15/5),l(c,25/10),l(d,40/20)]) means that a is a branch with leaves b,c,d, and its F is 15 and G = 0.

Goal (2) is unified with the 3 clause of expand that selects the subtree with the smallest f-value. In our case, this is b. However ths bound in this case is 25, the f-value of the second subtree of a. The next call to expand is

(3) expand([a], l(b,15/5), 25, _324,_325, Sol).

This goal is matched with the second clause of expand that finds the chiildren of b different of a.

The next call is

(4) expands([a],t(b,45/5,[l(e,45/10),l(f(45/15)]),25, _324,_325,Sol)

We notice that the F-value of b is 45, the smallest of the F's of its children.

This goal is satisfied by the 5-th clause of expand.

(5) expands([a],t(b,45/5,[l(e,45/10),l(f(45/15)]),25, t(b,45/5,[l(e,45/10),l(f(45/15)]), no,Sol)

Then goal (3) is satisfied by the instantiations of _324 and _325. We get

(6) expand([a], l(b,15/5), 25, t(b,45/5,[l(e,45/10),l(f(45/15)]), no, Sol).