32002: AI (Heuristic Search) **Spring 2014**

Lecture 27-29: March 20, 2014

Instructor: K.R. Chowdhary *: Professor of CS*

27.1 Heuristic Approach

The search efficiency can improve tremendously - reducing the search space, if there is a way to order the nodes to be visited in such that most promising nodes are explored first. These approaches are called *informed* methods, in contrast to the earlier uninformed or blind methods. These methods depend on some heuristics determined by the nature of the problem. The heuristics is defined in the form of a function, say f, which some how represents the mapping to the total distance between start node and the goal node. For any given node n, the total distance between start and goal node is $f(n)$, such that $f(n) = g(n) + h(n)$, here $g(n)$ is distance between start node and the node n, and $h(n)$ is the distance between node n and the goal node. We note that $g(n)$ can be easily determined and can be taken as shortest. However, the distance to goal is $f(x)$, which requires computation of $h(n)$, called heuristics, cannot be so easily determined. In deciding the next state, which is represented by node n, the state is chosen such that $f(n)$ is minimum.

Considering the case of the traveling salesman problem, which otherwise, is a combinatorially explosive problem, with exponential time complexity of $O(n!)$ for n nodes, reduces to only $O(n^2)$ if every time the next node selected is the nearest neighbor, that is, the one having shortest distance from the current node.

Similarly, in the 8-puzzle problem, the next move is chosen the one having minimum disagreement from the goal, i.e., having minimum number of misplaced positions with respect to the goal.

The heuristic methods, reduce the state space to be searched, and supposed to give the solution, but my fail also.

27.2 Hill-Climbing Methods

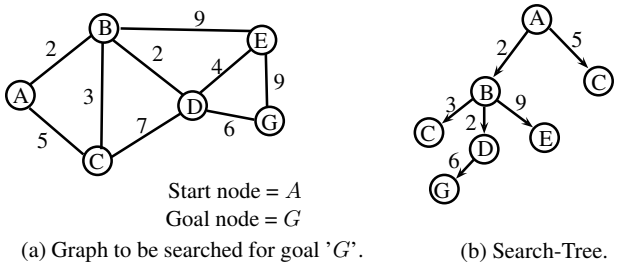
The hill-climbing algorithm 1 is an improved variant of the depth-first search method. A Hill-climbing method is called *greedy local search*. Local, because it considers a node close to the current node, at a time; and greedy because it selects nearest neighbor without knowing its future consequences. The inputs to this algorithm are **G**, **S**, and **Goal**, which stand for - graph, start(root) node, and the goal node, respectively.

Consider the graph shown in figure 27.1(a), where start node is A and goal node is G. It is required to find out the shortest path from node A to node G, using the method of hill-climbing. Figure 27.1(b) shows the search-tree for reaching to goal node G from start node A, with shortest path A, B, D, G and path length 10. We note that this path is in fact shortest, but it can be easily worked out that this approach cannot lead to shortest always.

Though Simple, hill climbing suffers from various problems.

Algorithm 1 Hill-Climb(Input: **G**, **s**, **Goal**)

```
1: Open = [s]
2: Closed = nil
3: if Open = nil then
4:    return fail
5: end if
6: repeat
7:    if Open.Head = Goal then
8:       return success
9:    end if
10:   expand Open.Head and generate children's set, call it C
11:   reject all paths in C having loops
12:   delete Open.Head and insert it into Closed
13:   sort C in order of heuristic, with best heuristic node is in its front
14:   insert C at the front of List
15: until Open = nil
16: return fail
```



(a) Graph to be searched for goal 'G'. (b) Search-Tree.

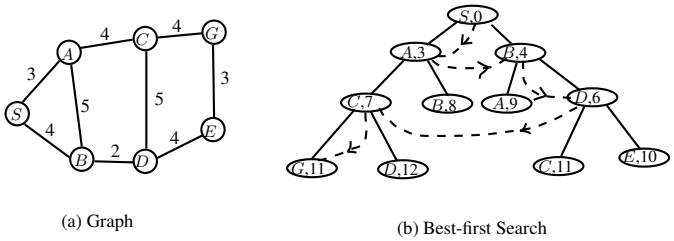
Figure 27.1: Graph with Hill-climbing search.

- **Foothill Problem:** This occurs when there are secondary peaks or *local maxima*, These are mistaken for the global maxima, as user is left with false sense of achieving the goal.
- **Plateau Problem:** This occurs when there is a flat region separating the peaks.
- **Ridge Problem:** It is like a knife edge or an edge on top of a hill, both the sides are valleys. It again gives a false sense of top of the hill, as no slope change appears.

Thus, hill climbing algorithms are *not complete.*

27.3 Best-First Search

If a problem has a very large search space and can be solved by iteration (unlike problems such as theorem proving where the search must continue until a proof is found), there is usually no alternative to using the iterative approach. Here, there is a serious problem in bounding the effort so that the search is tractable. For this reason, the search is usually limited in some way (e.g., number of nodes to be expanded, or maximum depth to which it may go). Since it is not expected that a goal node will be encountered, an evaluation function must be invoked to decide the approximate closeness to the goal for a given node at the periphery



(a) Graph

(b) Best-first Search

Figure 27.2: Best-First (branch-and-bound) Search.

of the search. This or a similar function can also be used for deciding which tip node to sprout from next. Thus evaluation functions and effort limits appear to be necessary for finding a solution by iteration.

Given a weighted directional graph $G = (V, E, W$ with a distinguished start node **s** and a set of goal nodes r, the optimal path problem is to find a least-cost path from **s** to any member of r where the cost of the path may, in general, be an arbitrary function of then, weights assigned to the nodes and branches along that path. A general best-first (GBF) strategy will pursue this problem by constructing a tree T of selected paths of G using the elementary operation of node expansion, that is, generating all successors of a given node. Starting with **S**, GBF will select for expansion that leaf node of T that features the highest "merit," and will maintain in T all previously encountered paths that still appear as viable candidates for *sprouting* an optimal solution path. The search terminates when no such candidate is available for further expansion, in which case the best solution path found so far is issued as a solution; if none has been found, a failure is proclaimed.

27.3.1 GBFS Algorithm

Since, best node is selected every time, it is guaranteed to give the best solution. The value of heuristic function $f(n)$ for a given node n, is not the indicative of distance from current node to the goal node, but since the best path is chosen every time, it is likely to provide the optimum solution for the problem.

Figure 27.2(a) shows the graph and a search tree for reaching goal node g from the start node **s** is shown in 27.2(b). Every node in best-first search shows the node identification along with its distance from the start node. The order in which the nodes are explored is shown with dotted line.

The algorithm 2 shows the steps for best-first search.

27.3.2 Analysis of Best-first search

Best-first searches tend to put the searching effort into those sub-trees that seem most promising (i.e. have the most likelihood of containing the solution). However, best-first searches require a great deal of bookkeeping for keeping track of all competing nodes, contrary to the great efficiencies possible in depth-first searches.

Depth-first searches, on the other hand, tend to be forced to stop at inappropriate moments thus giving rise to the horizon effect (number of possible states is immense and only a small portion can be searched). They also tend to investigate huge trees, large parts of which have nothing to do with any solution (since every potential arc of the losing side must be refuted). However, these large trees some-times turn up something that the evaluation functions would not have found were they guiding the search. Some times the efficiencies and discovery potential of the depth-first methods appear to out-weight what best-first methods have to offer. In fact, both methods have some glaring deficiencies.

Algorithm 2 Best-first Search(Input: **s**, **Goal**)

```
1: Open = [s]
2: Closed = nil
3: if Open = nil then
4:    return fail
5: end if
6: Insert start node s (i.e. root) at the begin of Open
7: repeat
8:    if Open.Head = Goal then
9:       return success
10:   end if
11:   generate children's set C of Open.Head
12:   rejects the new path with loops
13:   add C to either side of Open and record their parents
14:   if two or more paths reach a common node, delete all paths but shortest
15:   update distance from root for all C nodes
16:   delete Open.Head and insert into Closed
17:   sort Open by path length so that least cost path node is at front
18: until Open = nil
19: return fail
```

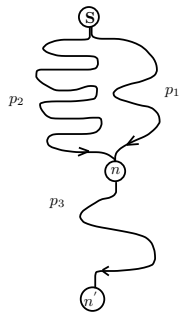


Figure 27.3: Order Preserving in GBFS.

In practice, several short-cuts have been devised to simplify the computation of GBF. First, if the evaluation function used for node selection always provides optimistic estimates of the final costs of the candidate paths evaluated, then we can terminate the search as soon as the first goal node is selected for expansion, without compromising the optimality of the solution used. This guarantee is called *admissibility* and is, in fact, the basis of the branch-and-bound method. This we can observe, for example in a chess game, where goal is property of a configuration and not the property of path from start node. hence, once a winning configuration is reached, there is no need to try it from other paths.

Second, we are often able to purge from T, large sets of paths that are recognized at an early stage to be dominated (i.e., superior) by other paths in T. This becomes particularly easy if the evaluation function f is *order preserving*, that is, if, for any two paths p_1 and p_2, leading from **s** to n, and for any common extension p_3 of those paths, the following holds (figure 27.3):

$$f(p_1) \geq f(p_2) \Rightarrow f(p_1 p_3) \geq f(p_2 p_3) \tag{27.1}$$

Order preservation is a judgmental version of the so-called principle of optimality in Dynamic Programming,

and it simply states that, if p_1, is judged to be more meritorious than p_2, both going from **s** to n, then no common extension of p_1 and p_2 may later reverse this judgment. Under such conditions, there is no need to keep in T multiple copies of nodes. Each time the expansion process generates a node n that already resides in T, we maintain only the lower-path to it, discarding the link from the more expensive father of n.

The best-first search allows revisiting the decisions. This is possible when a newly generated state by expansion of one of the state in open list is found in the closed list also. The best-first would retain the shorter path to this node, and purge the other to save space. However, in a variant of best-first, called, *greedy best-first* search once a state is visited the decision is final and the state is not visited again, thus eventually accepting the suboptimal solution. This however, does not require the *close* list, thus saving the memory space significantly.

Individual best-first search algorithms differ primarily in the cost function $f(n)$. If $f(n)$ is the depth of node n (not the distance from start), best-first search becomes breadth-first search. Note that breadth-first searches all the closer nodes (to start) before farther nodes. If $f(n) = g(n)$, where $g(n)$ is the cost of the current path from the start state to node n, then best-first search becomes Dijkstra's single-source shortest-path algorithm. If $f(n) = g(n) + h(n)$, where $h(n)$ is a heuristic estimate of the cost of reaching a goal from node n, then best-first search becomes the A^* algorithm.

Breadth-first search can terminate as soon as a goal node is generated, while Dijkstra's algorithm and A^* must wait until a goal node is chosen for expansion to guarantee *optimality*. Every node generated by a best-first search is stored in either the Open or Closed lists, for two reasons:

1. To detect when the same state has previously been generated. This is to prevent expanding it more than once.
2. To generate the solution path once a goal is reached. This is done by saving with each node a pointer to its parent node along an optimal path to the node, and then tracing these pointers back from the goal state to the initial state.

The primary drawback of best-first search is its memory requirements. By storing all nodes generated, best-first search typically exhausts the available memory in very short time on most machines.

While breadth-first search manages the Open list as a first-in first-out queue, general best-first searches manage the Open list as a *priority-queue* in order to facilitate efficiently determining the best node to expand next. All of these algorithms suffer the same memory limitation as breadth-first search, since they store all nodes generated in their Open or Closed lists, and will exhaust the available memory in a very short time.

27.4 A^* Search

By far, the most studied version of best-first-search is the algorithm A^*, which was developed for additive cost measures, that is, where the cost of a path is defined as the sum of the costs of its arcs. To match this cost measure, A^* employs an additive evaluation function $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of the currently evaluated path from s to n and h is a heuristic estimate of the cost of the path remaining between n and some goal node. Since $g(n)$ is *order preserving* and $h(n)$ depends only on the description of the node n, therefore $f(n)$ is also order preserving, and one is justified in discarding all but one parent for each node.

The figure 27.4 shows a graph, heuristic function table for $h(n)$ for every node in the graph, and tree constructed for A^* search for the graph, for given start state S and goal state G. To expand the next state (node), the one having smallest value of function f is chosen out of the nodes in the frontiers. The function

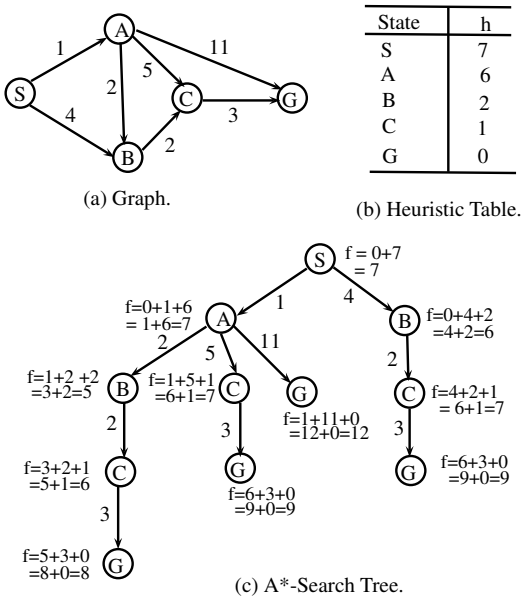


Figure 27.4: Graph and A*-Search-tree.

f for a node n is sum of three values: the g value of the parent of n, the distance from parent of n to the node n, and heuristic value (estimated distance from n to goal, given in the table) indicated by h. In the case of a tie, i.e., two states having equal values of f, the one to the left of the tree is chosen.

If, in addition, $h(n)$ is a lower bound to the cost of any continuation path from n to goal, then $f(n)$ is an optimistic estimate of all possible solutions containing the currently evaluated path. Then, terminating A^* upon the selection of the first goal node does not compromise its admissibility. Several other properties of A^* can be established if admissibility holds, such as the conditions for node expansion, node reopening, and the fact that the number of nodes expanded decreases with increasing h.

Based on the criteria set for A^* (i.e., to always expand the node having smallest value of f), the order of nodes expanded for figure 27.4(a), and shown in search-tree figure 27.4(c) with start node S and goal node G are: $(G, 0)$, $(B, 6)$, $(A, 7)$, $(B, 5)$, $(C, 6)$,$(C, 7)$ (with parent A), $(C, 7)$ (with parent B), $(G, 8)$, $(G, 9)$, $(G, 12)$. Finally, we note that the best path is corresponding to goal $(G, 8)$, and it is: S, A, B, C, G. Note that, in the A^*-tree, we followed the sequence $(S, 0)$, $(B, 6)$, with $(A, 7)$ and not $(C, 7)$, which are equally weighted. We chose the node to the left side subtree. Had we chosen, $(C, 7)$ in place of $(A, 7)$, we would have reached to $(G, 10)$ as next node, which incidentally, was not a good choice.

References

[1] Chowdhary K.R. (2020) Logic and Reasoning Patterns. In: Fundamentals of Artificial Intelligence. Springer, New Delhi. https://doi.org/10.1007/978-81-322-3972-7_9