Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*

15.1   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. 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,

$$f(n) = g(n) + h(n). \tag{15.1}$$

where $g(n)$ 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. The example for h is, 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.

15.2   Hill-Climbing Methods

The name hill-climbing comes from the fact that to reach to the top of a hill, one selects the steepest path at every node, out of the number of alternatives available. Naturally, one has to sort the slope values available, pick up the direction of move having highest angle, then reach to the next point (node) toward the hill top, then repeat the process. The hill-climbing algorithm 1 is an improved variant of the depth-first search method.

Consider the graph shown in figure 15.1(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 15.1(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.

Hill climbing suffers from various problems:

1. Rotating the brightness knob in control panel of an analog TV does not improve the quality of picture,

2. While testing a program, running it again and again, with different data sets does not indicate new discovery of errors,

---

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 in the front
14:     **insert C** at the front of **List**
15: **until Open** = nil
16: Return $fail$

Start node = A
Goal node = G

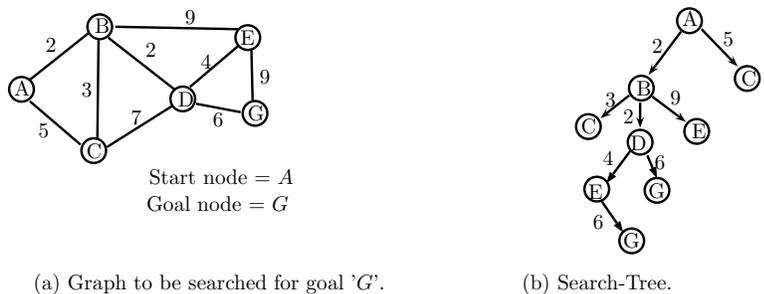(a) Graph to be searched for goal 'G'.   (b) Search-Tree.

Figure 15.1: Graph with Hill-climbing search.

3. Participating in a sports again and again (without new ideas and training), does not improve further performance.

Local Vs. Global Search

The preference for local search where only one state is considered at a time (out of the newly explored states), is good for a memory efficiency, compared to expanding all the successor nodes, but appear to be a too extreme step. Thus, instead, k best nodes out of successors generated are considered for further expansion. This gives rise to a new method, called *local beam search*.

However, the local beam search too will lead to concentration to a specific direction to those successors which generate more potential nodes. Consequently, this search ultimately becomes a local search. A better solution is to elect these k nodes such that they are not the best successors, but randomly selected, out of the next generation of successor nodes. This new method is called *stochastic beam search*.

15.3 Best-First Search

The heuristic approach typically uses special knowledge about the domain of the problem being represented by the graph to improve the computational efficiency of solution to particular graph-searching problem. However, the procedures developed via the heuristic approach generally have not been able to guarantee that minimum cost solution paths will always be found.

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 them, 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. Due to its nature of search, the best-first search is also called *branch-and-bound method*, i.e., branching a search to other direction to which path cost is minimum, and bounding the cost to that minimum, until a better minimum is found after next expansion.

15.3.1 GBFS Algorithm

A best-first search algorithm maintains two lists of nodes, an 'Open-list'and a 'Closed-list'. The Closed-list contains those nodes that have been expanded, by generating all their children, and the Open-list contains those nodes that have been generated, but not yet expanded.

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, does not here include the the distance from current node to the goal node, as required in equation (15.1, page no. 15-1), however, since the best path is chosen every time, it is likely to provide the optimum solution for the problem.

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

Example 15.1 *Best-first search.*

Figure 15.2(a) shows the graph, and figure 15.2(b) shows the search tree using GBFS for reaching to goal node G. The order in which the nodes are explored is shown with dotted line. Since G is goal, its path from root is S, A, C, G with shortest path length 11. Note that any other path will be of longer or equal length.
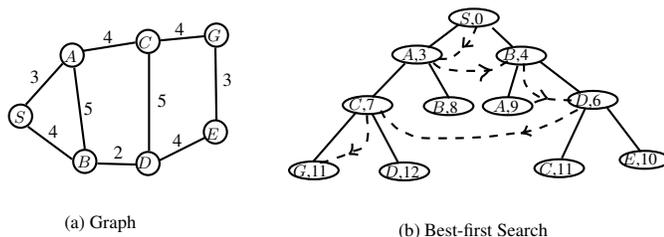

(a) Graph (b) Best-first Search

Figure 15.2: Best-First (Branch-and-Bound) Search.

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

 1: **Open** = $[S]$
 2: **Closed** = $[]$
 3: **repeat**
 4: 　　**if Open**.$Head$ = **Goal then**
 5: 　　　　return *success*
 6: 　　**end if**
 7: 　　generate children's set **C** of **Open**.$Head$
 8: 　　**if** $n \in C$ already exists in **OPEN** and new n is reachable by shorter path **then**
 9: 　　　　remove the old n
10: 　　**end if**
11: 　　**if** $n \in C$ already exists in **Closed** and reachable by shorter path **then**
12: 　　　　replace $n \in C$ by the same node from **Closed**, along with shorter distance from root
13: 　　**end if**
14: 　　**remove Open**.$Head$ and **insert** into **Closed**
15: 　　**update** distance from root for all **C** nodes
16: 　　add all **C** to either side of **Open** and record their parents
17: 　　sort **Open** by path length so that least cost path node is at front
18: **until Open** = nil
19: return *fail*

15.3.2　Analysis of Best-first search

If the evaluation function $f(n)$ 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*.

Second, we are often able to purge from tree 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 15.3):

$$f(p_1) \geq f(p_2) \Rightarrow f(p_1 p_3) \geq f(p_2 p_3) \tag{15.2}$$

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. This has been illustrated in figure 15.3.

If $f(n)$ is the total 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
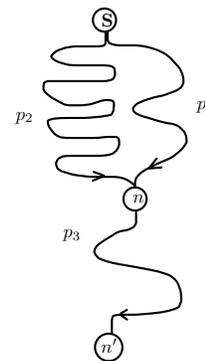

Figure 15.3: Order Preserving in GBFS.

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 a new
algorithm 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*.

15.3.3 Search Algorithm A^*

We can define the admissible search algorithm 3 for A^* (A-star).

Algorithm 3 Admissible-search A^*(Input: **G**, **S**, **Goal**)

1: **Open** $= [S]$
2: **Closed** $= []$
3: **repeat**
4:     compute $f^*(S)$
5:     Select the open node n whose $f^*(n)$ is smallest
6:     Resolve ties arbitrarily, but always in favor of any node $n \in$ **Goal**
7:     **if** $n \in$ **Goal then**
8:         move n to **Closed**
9:         terminate algorithm
10:    **else**
11:        insert n to **Closed**
12:        apply successor operator to n
13:        calculate f^* for each successor n_i of n
14:        insert all $n_i \notin$ **Closed** to **Open**
15:        move to **Open** any n_i in **Closed** and for which $f^*(n_i)$ is smaller now than it was when n_i was in
           Closed
16:    **end if**
17: **until Open** $= nil$
18: return *fail*

Example 15.2 A^*-*Search.*

The figure 15.4 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, the
one having smallest value of function f is chosen out of the nodes in the frontiers. The function 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.

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 15.4(a), and shown in search-tree in figure 15.4(c) with start node S and goal
node G are: $(G, 0)$, $(B, 5)$, $(A, 7)$, $(B, 5)$, $(C, 6)$,$(C, 7)$ (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 $(G, 9)$ as next node, which incidentally, was not a good choice.

15.3.4   Analysis of A^* Search

Let $h^*(n)$ be the *optimal cost* of the path from n to the goal node, and $g^*(n)$ is the optimal cost from start node to the node n, then $f^*(n)$ is *optimal* cost of path constrained to go through node n. The function $h^*(n)$ is undefined for node n that has no accessibility to goal node. Since, we intend to compute estimated minimum costs, let the corresponding figures for estimated costs are, $f(n)$, $g(n)$, and $h(n)$.

Assume that cost from start node **S** to node n is $c(s,n)$. This is equivalent to a function g^* as

$$g^*(n) = c(s,n) \qquad (15.3)$$

for all n accessible to **s**. Since $f^*(n)$ is optimal cost from start node **s** to goal node for any node n, we represent evaluation function $f^*(n)$ as

$$f^*(n) = g^*(n) + h^*(n) \qquad (15.4)$$

Here $f^*(n)$ is the cost of an optimal path from **s** constrained to go through node n. In this case $f^*(s) = f(s)$ is the condition for actual cost of an unconstrained optimal path from **s** to goal node.

(a) Graph.

(b) Heuristic Table.
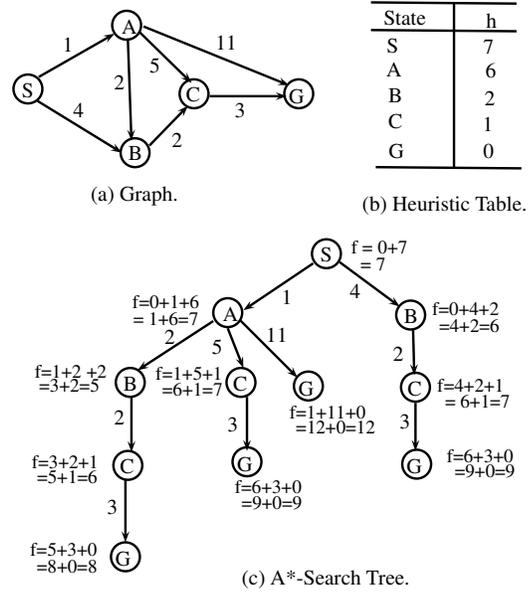
State	h
S	7
A	6
B	2
C	1
G	0

(c) A*-Search Tree.

Figure 15.4: Graph and A*-Search-tree.

Assume that f is an estimate of f^*. Then, we can extend the optimal functions definition for actual cost as,

$$f(n) = g(n) + h(n) \qquad (15.5)$$

where g and h are estimates of g^* and h^*, respectively. The $g(n)$ is cost of the path in the search tree from s to n computed by summing the arc costs encountered, while tracing the pointers back from n to s. This path is lowest cost path found so far by search algorithm. However, note that $g(n) \geq g^*(n)$.

The estimated value $h(n)$ for $h^*(n)$ can be obtained from the problem domain. For example, in the case of 8-puzzle problem or the 8-queen problem, this value is inverse of the number of tiles out of place with the goal or the inverse of number of queens giving checks to other queens. Even better value is, actual number of moves from n to goal.

Let us consider the specific cases of $f^*(n)$ as follows:

- When $h = 0$, then $g = d$ (the distance to goal in the search tree). This algorithm is called as **A**, and it is identical to breadth-first search (BFS).

- We claimed that the BFS algorithm is guaranteed to find the minimum path length to the goal node. If h is lower bound on h^*, i.e., $h(n) \leq h^*(n)$, for all n, then the algorithm will find an optimal path to a goal node. This algorithm is called as $\mathbf{A}^*$.