2.7 THE TRAVELLING SALESPERSON PROBLEM(TSP)

Is a touring problem in which each city must be visited exactly once. The aim is to find the shortest tour. The problem is known to be **NP-hard**. Enormous efforts have been expended to improve the capabilities of TSP algorithms. These algorithms are also used in tasks such as planning movements of **automatic circuit-board drills** and of **stocking machines** on shop floors.

VLSI layout

A **VLSI layout** problem requires positioning millions of components and connections on a chip to minimize area, minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield. The layout problem is split into two parts: **cell layout** and **channel routing**.

ROBOT navigation

ROBOT navigation is a generalization of the route-finding problem. Rather than a discrete set of routes, a robot can move in a continuous space with an infinite set of possible actions and states. For a circular Robot moving on a flat surface, the space is essentially two dimensional. When the robot has arms and legs or wheels that also must be controlled, the search space becomes multi-dimensional. Advanced techniques are required to make the search space finite.

2.8 AUTOMATIC ASSEMBLY SEQUENCING

The example includes assembly of intricate objects such as electric motors. The aim in assembly problems is to find the order in which to assemble the parts of some objects. If the wrong order is choosen, there will be no way to add some part later without undoing some work already done. Another important assembly problem is protein design, in which the goal is to find a sequence of Amino acids that will be fold into a three-dimensional protein with theright properties to cure some disease.

2.9 INTERNET SEARCHING

In recent years there has been increased demand for software robots that perform Internet searching, looking for answers to questions, for related information, or for shopping deals. The searching techniques consider internet as a graph of nodes(pages) connected by links.

2.10 UNINFORMED SEARCH STRATGES

Uninformed Search Strategies have no additional information about states beyond that provided in the **problem definition**.

Strategies that know whether one non goal state is "more promising" than another are called

Informed search or heuristic search strategies.

There are five uninformed search strategies as given below.

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

Breadth-first search

- Breadth-first search is a simple strategy in which the root node is expanded first, then all successors of the root node are expanded next, then their successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.

- Breath-first-search is implemented by calling TREE-SEARCH with an empty fringe that is a first-in-first-out (FIFO) queue, assuring that the nodes that are visited first will be expanded first. In otherwards, calling TREE-SEARCH (problem, FIFO-QUEUE()) results in breadth-first-search. The FIFO queue puts all newly generated successors at the end of the queue, which means that Shallow nodes are expanded before deeper nodes.

Figure 2.5 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

Properties of breadth-first-search

Time complexity for BFS

Assume every state has b successors. The root of the search tree generates b nodes at the first level, each of which generates b more nodes, for a total of b^2at the second level. Each of these generates b more nodes, yielding b^3nodes at the third level, and so on. Now suppose, that the solution is at depth d. In the worst case, we would expand all but the last node at level d, generating b^{d+1}- b nodes at level d+1.

Then the total number of nodes generated is $b + b^2 + b^3 + \ldots + b^d + (b^{d+1} + b) = O(b^{d+1)}$.

Every node that is generated must remain in memory, because it is either part of the fringe or is an ancestor of a fringe node. The space compleity is, therefore, the same as the time complexity

2.11 UNIFORM-COST SEARCH

Instead of expanding the shallowest node, **uniform-cost search** expands the node n with the lowest path cost. Uniform-cost search does not care about the number of steps a path has, but only about their total cost.

2.12 DEPTH-FIRST-SEARCH

Depth-first-search always expands the deepest node in the current fringe of the search tree. The progress of the search is illustrated in Figure 1.31. The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As those nodes are expanded, they are dropped from the fringe, so then the search "backs up" to the next shallowest node that still has unexplored successors.

This strategy can be implemented by TREE-SEARCH with a last-in-first-out (LIFO) queue, also known as a stack.

Depth-first-search has very modest memory requirements. It needsto store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path. Once the node has been expanded, it can be removed from the memory, as soon as its descendants have been fully explored (Refer Figure 2.7).

For a state space with a branching factor b and maximum depth m, depth-first-search requires storage of only bm + 1 nodes.

Using the same assumptions as Figure, and assuming that nodes at the same depth as the goal node have no successors, we find the depth-first-search would require 118 kilobytes instead of 10 petabytes, a factor of 10 billion times less space.

Drawback of Depth-first-search

The drawback of depth-first-search is that it can make a wrong choice and get stuck going down very long(or even infinite) path when a different choice would lead to solution near the root of the search tree. For example, depth-first-search will explore the entire left subtree even if node C is a goal node.

2.12 BACKTRACKING SEARCH

A variant of depth-first search called backtracking search uses less memory and only one successor is generated at a time rather than all successors.; Only O(m) memory is needed rather than O(bm)

DEPTH-LIMITED-SEARCH

The problem of unbounded trees can be alleviated by supplying depth-first-search with a pre- determined depth limit l. That is, nodes at depth l are treated as if they have no successors. This approach is called **depth-limited-search**. The depth limit soves the infinite path problem.

Depth limited search will be nonoptimal if we choose l > d. Its time complexity is $O(b^l)$ and its space complete is O(bl). Depth-first-search can be viewed as a special case of depth limited search with l = oo Sometimes, depth limits can be based on knowledge of the problem. For, example, on the map of Romania there are 20 cities. Therefore, we know that if there is a solution, it must be of length 19 at the longest, So l = 10 is a possible choice. However, it can be shown that any city can be reached from any other city in at most 9 steps. This number known as the **diameter** of the state space, gives us a better depth limit.

Depth-limited-search can be implemented as a simple modification to the general tree search algorithm or to the recursive depth-first-search algorithm. The pseudocode for recursive depth- limited-search is shown in Figure.

It can be noted that the above algorithm can terminate with two kinds of failure : the standard *failure* value indicates no solution; the *cutoff*value indicates no solution within the depth limit. Depth-limited search = depth-first search with depth limit l,returns cut off if any path is cut off by depth limit

```
function Depth-Limited-Search( problem, limit) returns a solution/fail/cutoff return
Recursive-DLS(Make-Node(Initial-State[problem]), problem, limit) function
Recursive DLS(node, problem, limit) returns solution/fail/cutoff cutoff-occurred?
false
if Goal-Test(problem,State[node]) then return Solution(node)
else if Depth[node] = limit then return cutoff
else for each successor in Expand(node, problem) do result
Recursive-DLS(successor, problem, limit) if result = cutoff then
cutoff_occurred?true else if result not = failure then return result
ifcutoff_occurred? then return cutoff else return failure
```

Figure 2.9 Recursive implementation of Depth-limited-search

2.13 ITERATIVE DEEPENING DEPTH-FIRST SEARCH

Iterative deepening search (or iterative-deepening-depth-first-search) is a general strategy often used in combination with depth-first-search, that finds the better depth limit. It does this by gradually increasing the limit – first 0,then 1,then 2, and so on – until a goal is found. This will occur when the depth limit reaches

d, the depth of the shallowest goal node. The algorithm is shown in Figure.

Iterative deepening combines the benefits of depth-first and breadth-first-search Like depth-first-search, its memory requirements are modest; O(bd) to be precise.

Like Breadth-first-search, it is complete when the branching factor is finite and optimal when the path cost is a non decreasing function of the depth of the node.

Figure shows the four iterations of ITERATIVE-DEEPENING_SEARCH on a binary search tree, where the solution is found on the fourth iteration.