CHAPTER – 2

PROBLEM SOLVING

Problem Space:

In AI to solve a problem by using a problem solving agent, there must be a problem space available. A problem space is actually the collection of initial situation of the problem (initial state), final solution (goal state) and all other possible alternatives to move from initial state to goal state.

In AI problem solving, problem formulation is the process of deciding what action and states to consider to achieve the goal. In AI a problem solution can be considered as searching a suitable path in a state space from initial state to goal state.

Problem Types:

1. Single State Problem:

This is the simplest problem type where there exist only one possible state for solving any problem.

2. Multiple State Problem:

Most of the problem in real world are multiple state problem where more than one state problem are available in state space search are analyze to find the appropriate solution.

3. Contingency Problem:

Many problem in the real world are contingency problem where exact prediction is impossible to move on from a particular state. Such problems require complex algorithm to determine the best suitable next state from a particular state.

4. Exploration Problem:

In this type of problem an agent has no information about the effects of its action and no-fixed states are available except the initial and goal states. In this case the agent must perform experiments to discover the actions, their effects to move on towards the goal states. In every step the problem solving agent explore the all possible alternatives.

Well Defined Problems:

A problem is really a collection of information that the agent will use to decide what to do while solving a problem. A problem is known as well-defined problem when it is defined by four basic components as:

1. Initial State:

The initial state is the state in which the agent initially exist or the state that is well known to the agent. Depending on the nature of the problem there may exist one or multiple initial state.

2. Operators:

Operators denotes actions necessary to generate next possible states. A successor function generally used as an operator to generate next step.

3. Goal Test:

The goal test is used to compare the current state with the goal state. A problem may have single or multiple goal state.

4. Path Cost:

A path cost function is a function that assigns a cost to the path from initial state to goal state. There may exist multiple paths from initial state to goal state and the agent must choose a path with minimum cost path.

In AI the solution of any problem is the optimum path from initial state to goal state satisfying the goal test function.

Example: Define the 8 puzzle problem as a well-defined problem.

1. Initial State:

A state that describes the specific locations of 8-digits (1 to 8) out of 9 squares, where one square tile is left as blank like:

3	5	2
1	7	
6	8	4

2. Operator:

The movement of blank tile either to left, right, up or down.

3. Goal Test:

The state that matches the goal state of the problem where the goal state may be as:

1	2	3
4	5	6
7	8	

4. Path Cost:

The path cost is the length of the path from initial state to goal state in the search tree, assuming that each step cost 1.

Measuring Performance of Search Strategies:

To evaluate the performance of search algorithms we use four criteria as:

1. Completeness:

A search algorithm is said to be complete, if it guarantees to find the solution of a problem if exist.

2. Optimality:

A search algorithm is said to be optimum, if it guarantees to find the highest quality solution (that means the solution with minimum cost), if there exist multiple solutions.

3. Time Complexity:

It means how long (in worse or average condition) does it take to find a solution? Usually it is measured in terms of **numbers of nodes expanded**.

4. Space Complexity:

It means how much space is used by the algorithm to provide the solution? Usually it is measured in terms of the **maximum numbers of nodes in a memory at a time.**

Search Strategies:

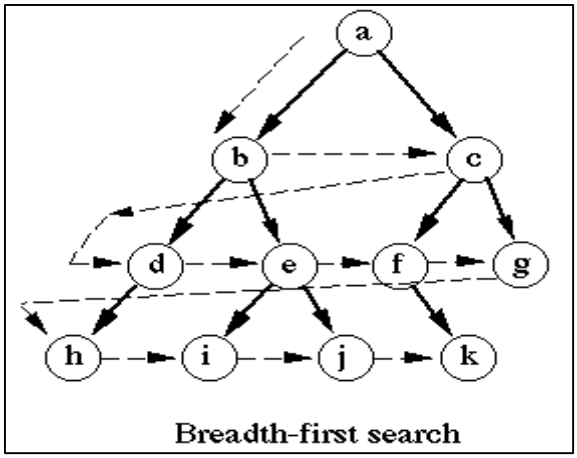
1. Blind or Uninformed Search:

In uninformed or blind search strategies no information other than initial state the operators and goal test function are available while solving any problem. A blind search should proceed in a systematic way by exploring nodes in some predetermined order or simply by selecting nodes at random. This approach does not provide path cost related information. Few commonly used blind search algorithms are:

a. Breadth First Search:

Breadth First Search is performed by exploring all the nodes at a given depth before proceeding to next level. An algorithm for BFS is quite simple and it use a queue structure to hold all generated but still unexplored nodes. The BFS algorithm proceeds as follows:

i. Place the starting node on the queue.
ii. If the queue is empty return failure and stop.
iii. If the first element on the queue is the goal node then return success and stop otherwise,
iv. Remove and expand the first element from the queue and place all the children at the queue in any order.
v. Return to step ii.



Breadth-first search

Analysis:

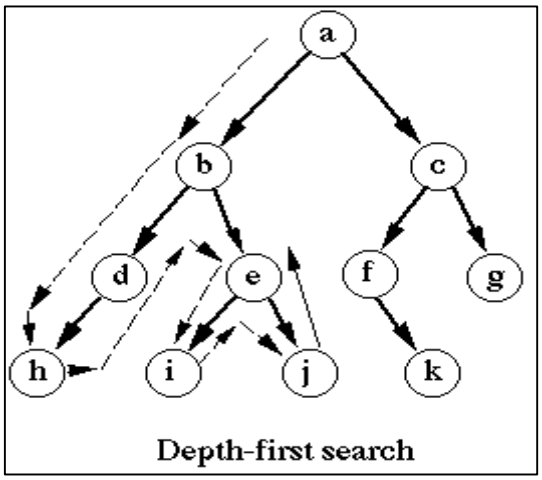
- **Completeness:** Yes

- **Optimality:** Yes
- **Time Complexity:** The maximum numbers of nodes generated, if the solution is available at depth 'd' is: $1 + b + b^2 + \cdots\cdots + b^d \equiv O(b^d)$
- **Space Complexity:** The maximum number of nodes that has to be stored in a memory at a time, is space complexity and in this case if the branching factor is 'b' and the solution is available at depth 'd' then space complexity is $O(b^d)$

b. Depth First Search:

Depth First Search are performed by diving downward into a tree as quickly as possible. To implement DFS, we use a stack (LIFO data structure) and the algorithm proceeds as follows:

i. Place the starting node on the stack.
ii. If the stack is empty, return failure and stop.
iii. If the top element in the stack is the goal node then return success and stop, otherwise,
iv. Remove the top element from the stack and expand it place all the children on the top of stack in any order.
v. Return to step ii.



Depth-first search

Analysis:

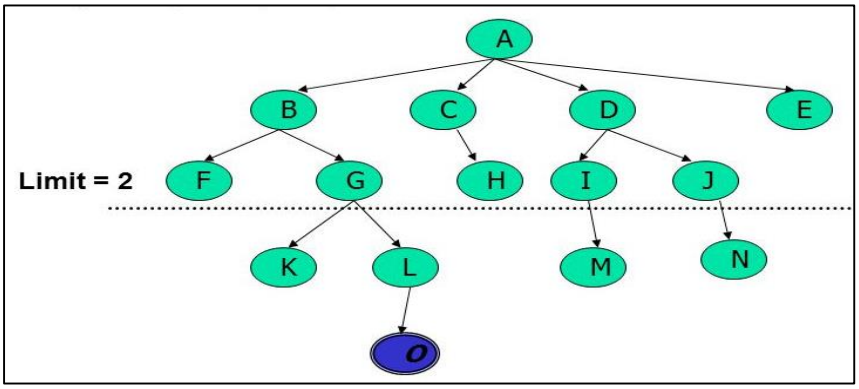
- **Completeness:** Yes, because it compares every node of the tree until goal is accomplished.
- **Optimality:** No, because this algorithm may generate a solution at the deepest node first even the solution may exist at the shorter part on the other side of the tree.
- **Time Complexity:** The maximum numbers of nodes generated, if the solution is available at depth 'd' is: $1 + b + b^2 + \cdots\cdots + b^d \equiv O(b^d)$
- **Space Complexity:** If the solution is available is available at depth 'd' then maximum numbers of nodes that have to be stored in memory at a time are $b + b + b + \cdots\cdots + b.d \equiv O(b.d)$, where b is the branching factor.

Branching factor 'b' of a tree indicates the maximum number of children of any node in the tree.

c. Depth Limited Search:

The unbounded tree problem appeared in DFS can be fixed by imposing a limit on the depth that DFS can reach, this limit we will call depth limit l, this solves the infinite path problem. DLS can be used when the there is a prior knowledge to the problem, which is always not the case,

typically, we will not know the depth of the shallowest goal of a problem unless we solved this problem before.



Analysis:

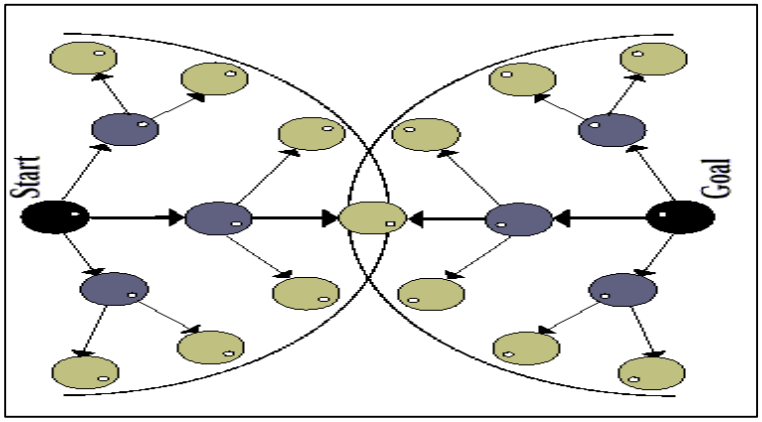
- **Completeness:** The limited path introduces another problem which is the case when we choose l < d, in which our DLS will never reach a goal, in this case we can say that DLS is not complete.
- **Optimality:** One can view DFS as a special case of the depth DLS, that DFS is DLS with l = infinity. DLS is not optimal even if l > d.
- **Time Complexity:** $O(b^l)$
- **Space Complexity:** $O(b * l)$

d. Bi-directional Search:

When a problem has a single goal state that is given explicitly and all node generation operators have inverse (that means the ability to generate previous states from the given state by applying operators), then bi-directional search can be used.

Bi-directional search is performed by searching forward from the initial node and backward from the goal node simultaneously. To do so the program must store the nodes generated on both search until a common node is found.

We can apply both BFS and DFS for bi-directional search if suitable operators are available.



Analysis:

- **Completeness: Yes**
- **Optimality: Yes**

- **Time Complexity:** $O\left(b^{d/2}\right)$
- **Space Complexity:** $O\left(b^{d/2}\right)$**,** if its implemented through BFS.

2. *Informed or Heuristic Search:*

When more information than the initial state, operators and goal test function is available, then the size of search space can usually be constrained that means size of search space can be drastically reduced to provide acceptable solutions in acceptable time such methods are known as informed search methods. These methods depend on the use of heuristic information for more efficient search process.

Information about the problem such as the nature of the state the cost of transforming from one state to another state the possibility of taking certain path, the characteristics of the goal, etc. can be useful to guide the search process more efficiently. This information known as heuristic information can be expressed in the form of heuristic function f(n). Few commonly used informed search strategies are:

a. Best First Search:

Best First Search depends on the use of heuristic information to select the most promising path to the goal node. This algorithm retains all estimates computed for previously generated nodes and makes its selection based on the best among all of them. So, at any point in the search process, it moves forward from the most promising of all the nodes generated so far.

Algorithm:

i. Place the starting node on the queue.
ii. If the queue is empty, return failure and stop.
iii. If the first element on the queue is the goal node then return success and stop. Otherwise,
iv. Remove the first element from the queue, expand it and compute the estimated cost for each child node. Place the children on the queue and arrange all queue elements in ascending order of cost so that the node with lowest cost is at the first position.
v. Return to step ii.

b. Hill Climbing Search:

In this method at each point in the search path, a successor node that appear to lead to the goal node (top of the hill) must quickly is selected for exploration. This method requires some information to evaluate the cost of successor node.

Hill climbing is similar to depth first searching, where the most promising child is selected for expansion and previous nodes or children are ignored in every step. The simplest way to implement hill climbing is as:

i. Evaluate the initial state. If it is a goal state then return it and stop. Otherwise continue with the initial state as the current state.
ii. Loop until a solution is found or there is no new update operator left to be applied in the current state.
    a. Select an operator that has not been applied to the current state and apply it to produce a new state.
    b. Evaluate the new state:

1. If it is a goal state then return it and stop.
2. If it is not a goal state but it is better than current state then make it the current state.
3. If it is not better than the current state then continue in the loop.

This algorithm has three basic problems and they are:

1. **Local Maximum:**

A local maximum is a state that is better than all its neighbor but is not better than some other state further away.

2. **Plateau:**

A plateau is a flat area of search space in which a whole set of neighboring states have the same value. On a plateau it is not possible to determine the best direction in which to move by making local comparisons.

3. **Ridge:**

It is a special kind of local maximum. It is an area of search space that is higher than surrounding area but does not leads to the goal point.

c. A* Search:

The A* algorithm is a specialization of best first search. It provides general guidelines to estimate goal distances in the search graph. The main point of A* search is the heuristic cost function which is defined as: $f*(n) = g*(n) + h*(n), where$

The component $g*(n)$ is the cost estimate from the starting node upto the current node "n". And the component $h*(n)$ is the cost estimates from the current node "n" upto the goal node. The algorithm proceeds as follows:

1. Place the starting node on a queue (OPEN).
2. It OPEN is empty then stop and return failure.
3. Remove a node "n" from OPEN that has the smallest value of $f*(n)$. If the node "n" is the goal node then return success and stop. Otherwise,
4. Expand node "n", generating all of its successors (n') and place "n" on queue CLOSED. For every successor (n'), if (n') is not already on OPEN, attach a back pointer to "n", compute $f*(n')$ and place one queue OPEN.
5. Return to step 2.

d. Greedy Search:

A greedy algorithm is an algorithmic paradigm that follows the problem solving heuristic of making the locally optimal choice at each stage with the intent of finding a global optimum. In many problems, a greedy strategy does not usually produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.

For example, a greedy strategy for the traveling salesman problem (which is of a high computational complexity) is the following heuristic: "At each step of the journey, visit the nearest unvisited city." This heuristic doesn't intend to find a best solution, but it terminates in a reasonable number of steps; finding an optimal solution to such a complex problem typically

requires unreasonably many steps. In mathematical optimization, greedy algorithms optimally solve combinatorial problems having the properties of Metroid, and give constant-factor approximations to optimization problems with sub-modular structure.

In general, greedy algorithms have five components:

1. A candidate set, from which a solution is created
2. A selection function, which chooses the best candidate to be added to the solution
3. A feasibility function, that is used to determine if a candidate can be used to contribute to a solution
4. An objective function, which assigns a value to a solution, or a partial solution, and
5. A solution function, which will indicate when we have discovered a complete solution