Problem Solving Agent (goal-based)

This chapter describes one kind of goal-based agent called a problem-solving agent. Our discussion of problem solving begins with precise **definition** of problems and their solutions and give several examples to illustrate these definitions. We then describe several **general-purpose search algorithms** that can be used to solve these problems. We will see several **uninformed** search algorithms, algorithms that are given no information about the problem other than its definition. Although some of these algorithms can solve any solvable problem, none of them can do so efficiently. **Informed** search algorithms, on the other hand, can quite will given some guidance on where to look for solutions.

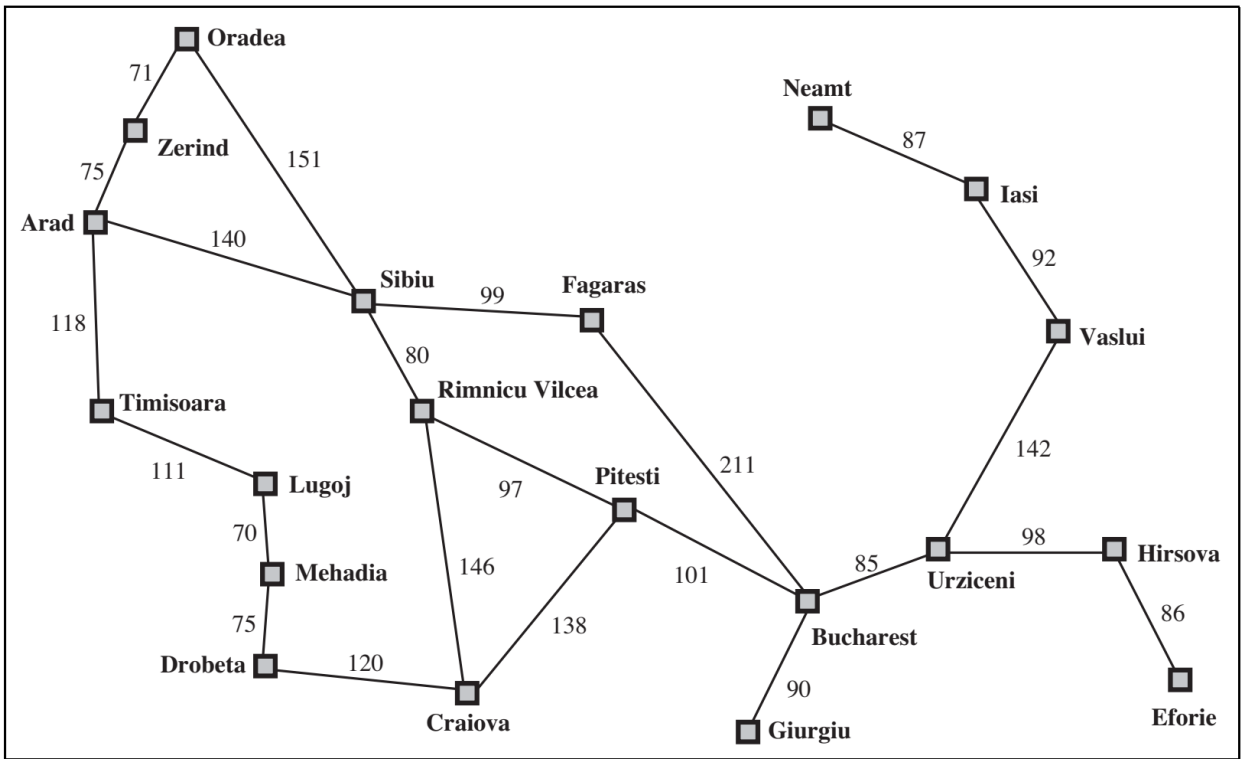
1. **Formulating problems**: a problem can be defined formally by five components:
    a. **The initial state** that agent starts in. For example, the initial state for our agent in Romania might be describe as ***In (Arad)***.
    b. A description of the possible **actions** available to the agent. Given a particular state (s), Actions(s) returns the set of actions that can be executed in s. we say that each of these actions is applicable in s. for example, from the state In (Arad), the applicable actions are { Go(Sibiu), Go(Timisoara), Go (Zerind)}.
    c. A description of what each action does; the formal name for this is the **transition model**. Specified by function Result (s,a) that returns the state results from doing action a in state s.

$$\text{Result}(\text{in}(\text{Arad}),\text{Go}(\text{Zerind})) = \text{In}(\text{Zerind})$$

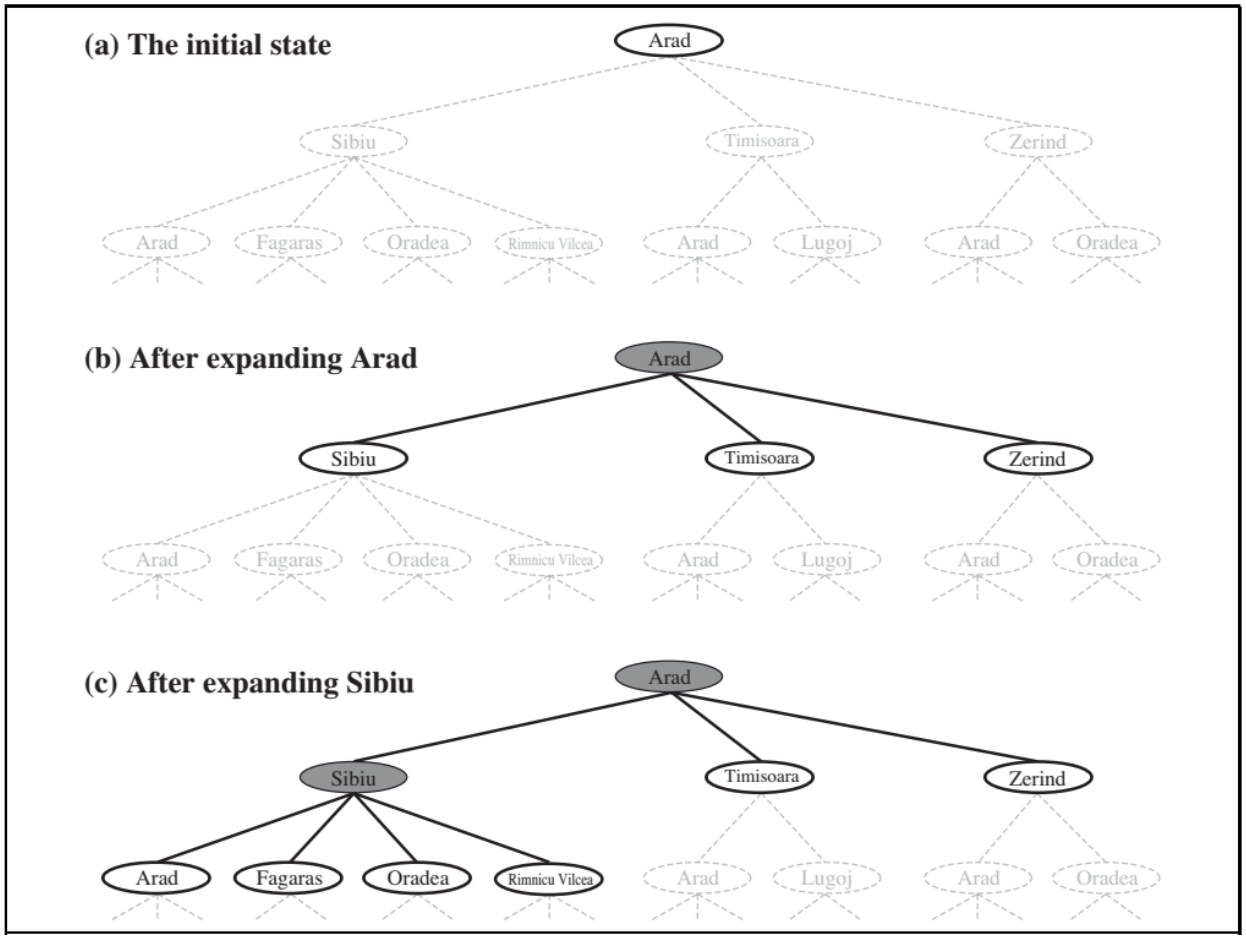
    d. The **goal test**, which determines whether a given state is a goal state. Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them.

E-mail: hasanein.y.m.alhrabi@uomus.edu.iq

e. A path **cost function** that assigns a numeric cost to each path. The problem-solving agent chooses a const function that reflects its own performance measures.



2. **Searching for Solutions:** having formulated some problems, we now need to solve them. A solution is an action sequence, so **search algorithms work by considering various possible actions sequences**. the possible action sequences starting at the initial state for a **search tree** with the initial state at the root, the branches are actions and the **nodes** correspond to states in the state space of the problem.



Measuring problem-solving performance: before we get into the design of specific search algorithms, we need to consider the criteria that might be to choose among them. We can evaluate an algorithm's performance in four ways:

1. Completeness: it the algorithm guaranteed to find a solution when there is one.
2. Optimality: does the strategy find the optimal solution. Solution quality is measured by the path cost function, and an optimal solution has the lowest path cost among all solutions.
3. Time complexity: how long does it take to find a solution.
4. Space complexity: how much memory is needed to perform the search.

E-mail: hasanein.y.m.alhrabi@uomus.edu.iq

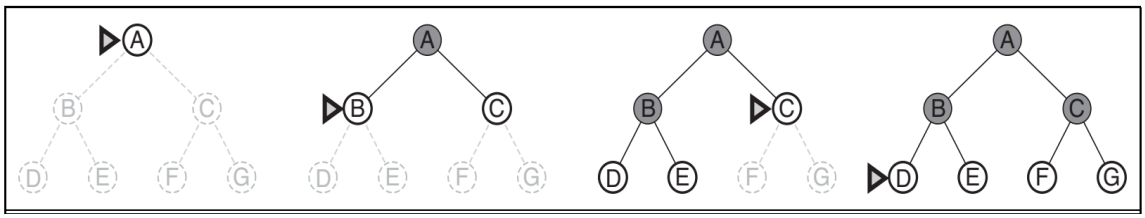
Search Strategies

This section covers several search strategies that come under the heading of :

1. Uninformed search (blind search): the term means the strategies have no additional information about the states beyond that provided in the problem definition. All they can do is generate successors and distinguish a goal state from non-goal state.
2. Informed search (heuristic) : one that uses problem-specific knowledge beyond the definition of the problem itself, can find solutions more efficiently than can an uninformed strategy.

Uninformed Search Strategies

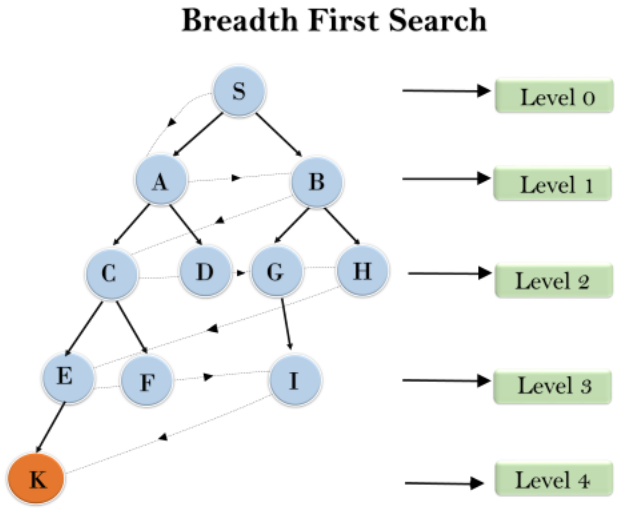
1. **Breadth-first search:** it is a simple strategy in which the **root node** is expanded first, then all the **successors** of the root node are expanded next, then their successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded. In Breadth-first search the shallowest unexpanded node is chosen for expansion. This is achieved very simply by using **FIFO** queue for the frontier. Thus, new nodes go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first. The **goal test** is applied to each node when its generated rather than when it is selected for expansion.



Example: in the following example we show the traversing of the tree using BFS algorithm form the root node S to goal node K.

E-mail: hasanein.y.m.alhrabi@uomus.edu.iq

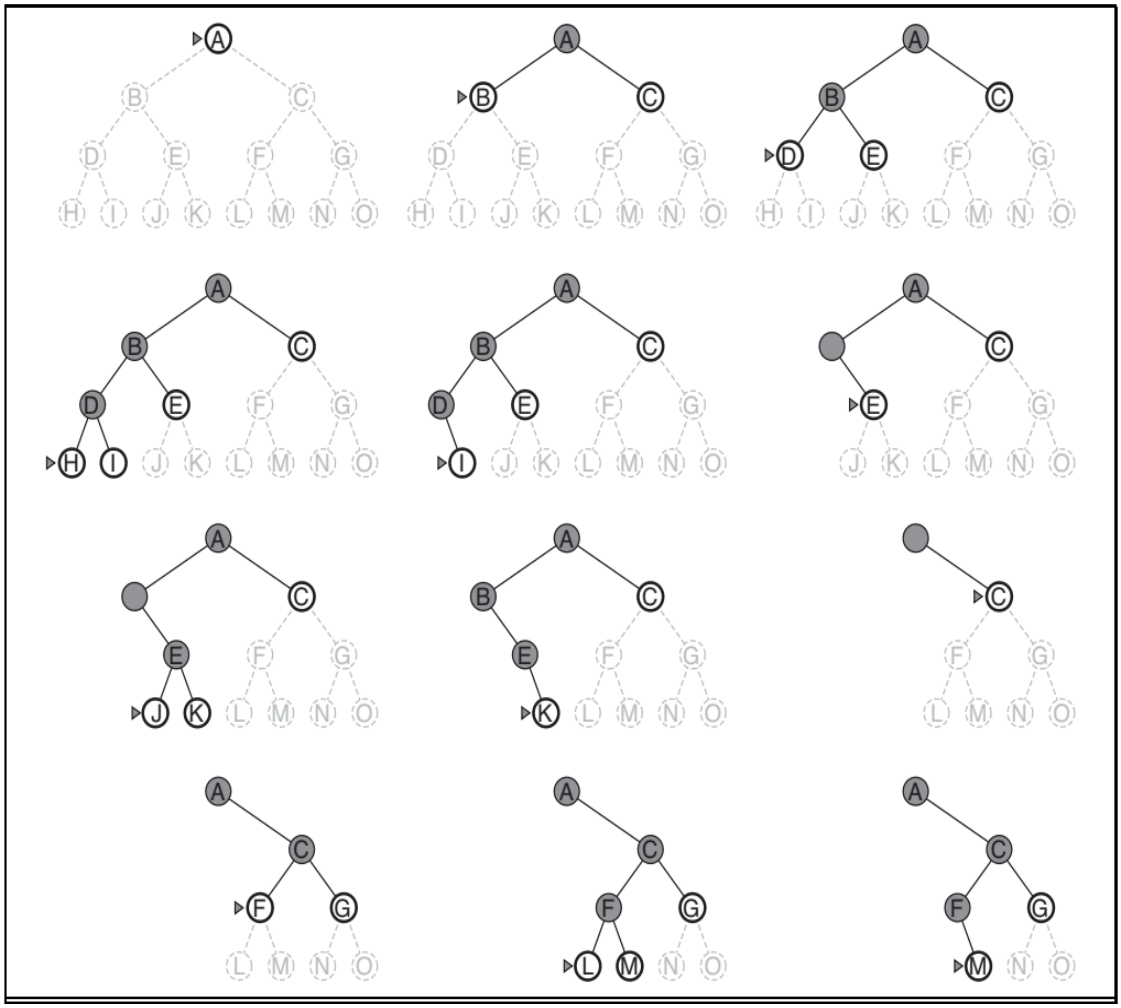
S---> A--->B---->C--->D---->G--->H--->E---->F---->I---->K



The breadth-first search according to the four performance criteria, we can easily see that:

- It is complete
- Optimal if all paths have the same cost.
- Time and space complexity is not so good.

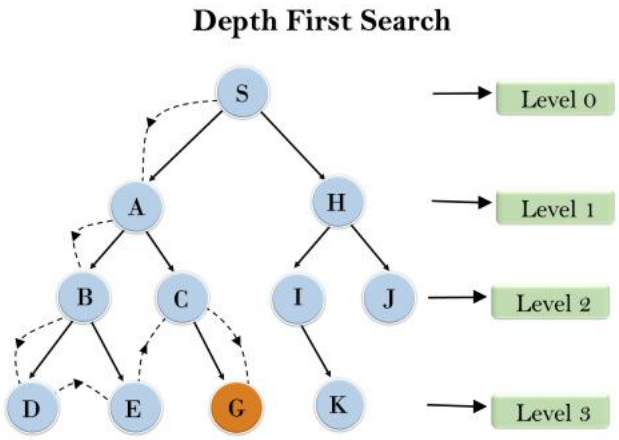
2. **Depth-first search:** it always expands the deepest node in the current frontier of the search tree. The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As those nodes are expanded, they are dropped for the frontier, so then the search backs up to the next deepest node that still has unexplored successors. Depth-first search uses a LIFO queue, A LIFO queue means that the most recently generated node is chosen for expansion. This must be the deepest unexpanded node because it is one deeper than its parent, which in turn was the deepest unexpanded node when it was selected. The goal test is applied when the node is selected for expansion.



Example: in the below search tree, we have shown the flow of depth-first search, and it will follow the order as

Root node ---------- > Left node ---------- > Right Node

E-mail: hasanein.y.m.alhrabi@uomus.edu.iq



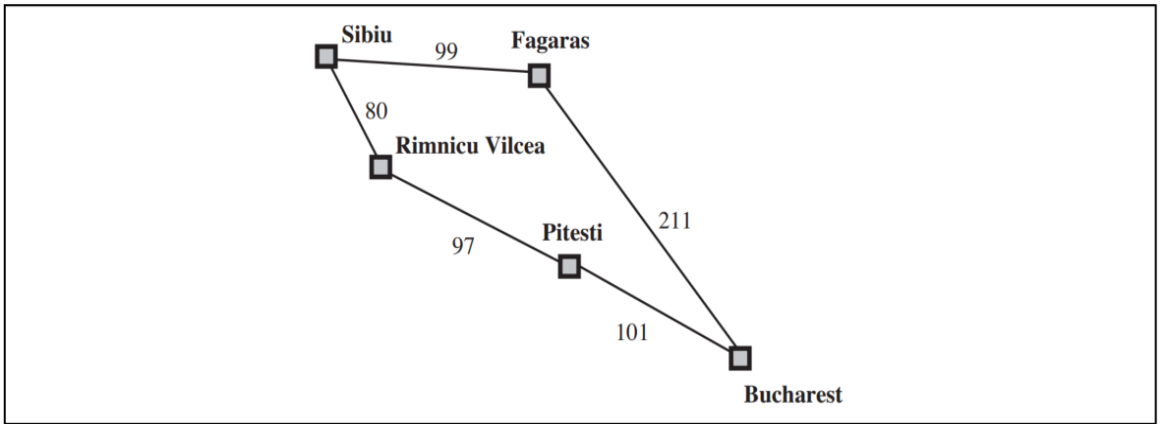
3. **Uniform-cost search:** when all steps cost are equal, breadth-first search is optimal because it always expands the shallowest unexpanded node. By a simple extension, we can find an algorithm that is optimal with any step-cost function. Instead of expanding the shallowest node, uniform-cost search expands the node n with the lowest path cost g(n). This is done by storing the frontier as a priority queue ordered by g.

   In addition to the ordering of the queue by path cost, there are two other significant differences from breadth-first search.

   First: the goal test is applied to a node when it is selected for expansion rather than when it is first generated.

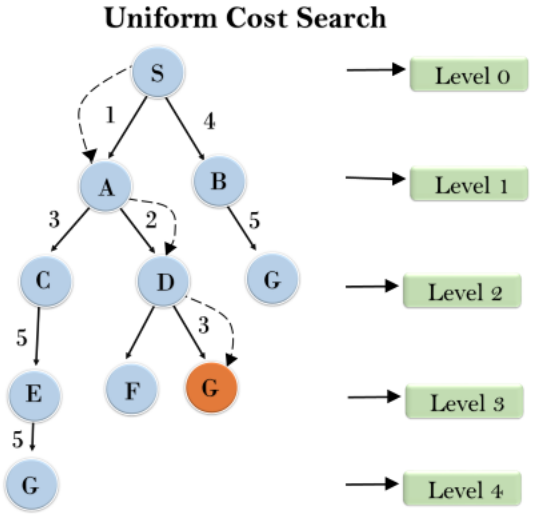
   Second: the second difference is that a test is added in case a better path is found to a node currently on the frontier.

   Both of these modifications come into play in the following example:

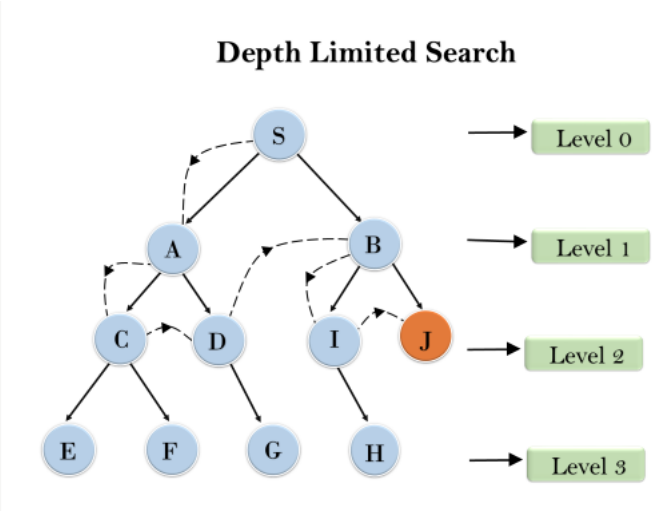


where the problem is to get is to get from Sibiu to Bucharest. The successors of Sibiu are Rimnicu Vilcea and Fagaras, with cost 80 and 99, respectively. The least-cost node, Rimnicu Vilcea, is expanded next, adding Pitesti with cost 80 + 97 = 177. The least-cost node is now Fagaras, so it is expanded, adding Bucharest with cost 99+211 = 310. Now a goal node has been generated, but uniform-cost search keeps going, choosing Pitesti for expansion and adding a second path to Bucharest with cost 80 + 97 + 101 = 278. Now the algorithm checks to see if this new path is better than the old one, it is, so the old one is discarded. Bucharest, now with g-cost 278, is selected for expansions and the solution is returned.

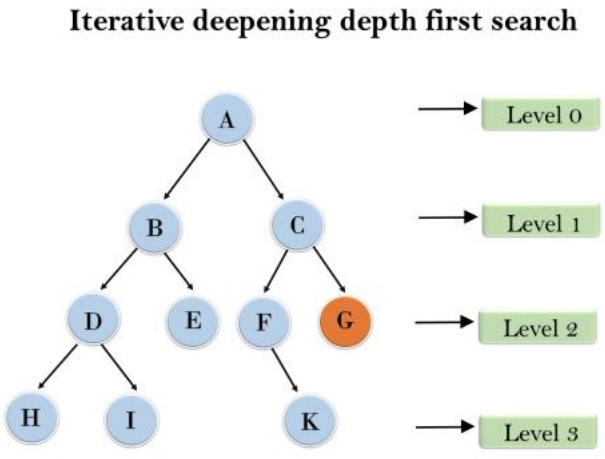
Example: the following example shows how to traverse a tree using uniform cost search.



4. **Depth-limited search:** the embarrassing failure of depth-first search in infinite state spaces can be alleviated by supplying depth-first search with a predetermined depth limit ***L*** **.** that is, nodes at depth ***L*** are treated as if the have no successors. The following examples explain the implementation of depth-limited search algorithm.



5. **Iterative deepening depth-first search:** the iterative deepening search is a general strategy, often used in combination with depth-first tree search, that finds the best depth limit. It does this by gradually increasing the limit, first 0, then 1, then 2 and so on, until a goal is found. The following example showing the iterative deepening depth-first search.



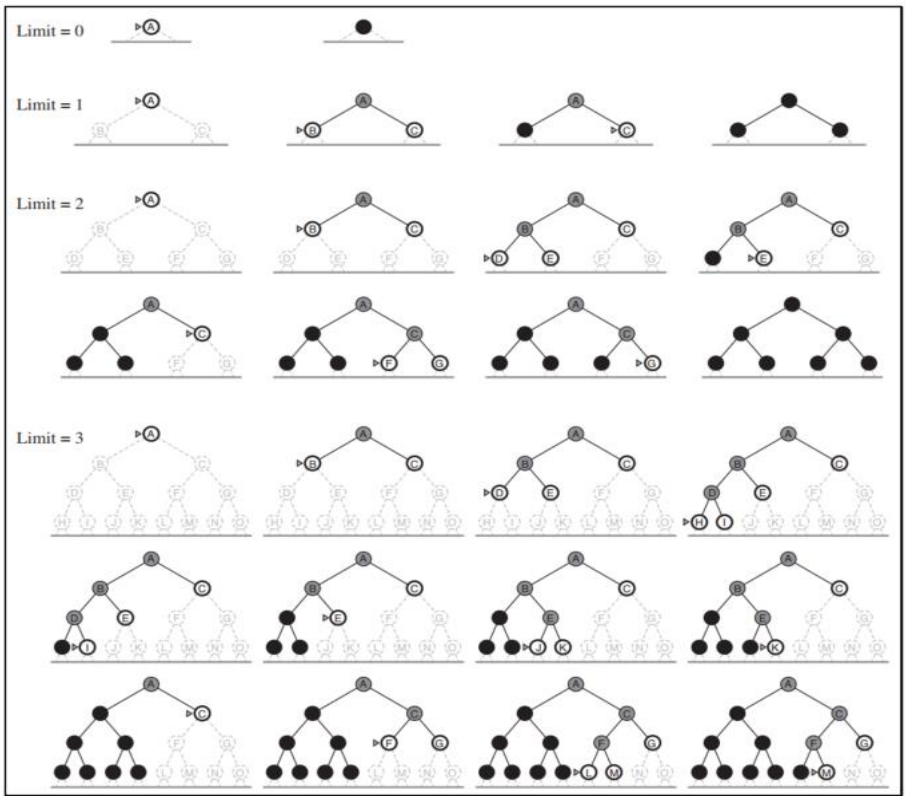
1st Iteration ---------- A

2nd iteration ---------- A, B,C

3rd iteration ---------- A,B,D,E,C,F,G

4th iteration ---------- A, B, D, H, I, E, F, K, G.

A more detailed example can be explained in the following figure.



6. **Bidirectional search:** the idea behind bidirectional search is to run two simultaneous searches, one forward from the initial state and the other backward from the goal, hoping that the two searches meet in the middle. Bidirectional search is implemented by replacing the goal test with a check to see whether the frontiers of the two searches intersect, if they do, a solution has been found. The check can be done when each node is generated or selected for expansion.