Solving Problems by Searching

Under the assumptions that, the solution to any problem is a fixed sequence of actions. For example, the agent might plan to drive from Baghdad to Basrah, then if the agent knows the initial state and the environment is known and deterministic (it knows exactly where it will be after the first action and whit it will perceive). Since only one percept is possible after the first action, the solution can specify only one possible second action, and so on.

The process of looking for a sequence of actions that reaches the goal is called **Search**. A search algorithm takes a **problem** as input and returns a **solution** in the form of **action sequence**. Once a solution is found, the actions it recommends can be carried out, this is called the **execution phase**. Thus, we have a simple "**formulate, search, execute**" design for the agent, as shown in figure below.

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
              state, some description of the current world state
              goal, a goal, initially null
              problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
      if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

Figure 3.1 A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.

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

After formulating a goal and a problem to solve, the agent calls a search procedure to solve it. It then uses the solution to guide its actions, doing whatever the solution recommends as the next thing to do. Once the solutions have been executed, the agent will formulate a new goal.

1. **Well defined problems and solutions:** a problem can be defined formally by five components:
   1. The initial state that the agent starts in. for example the initial state for an agent travel from Baghdad to Basrah is In (Baghdad).
   2. 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 see that each of these actions is applicable in s.
   3. A description of what each action does. The formal name for this is the **transition model**, specified by a function **RESULT**(s,a) that returns the state that results for doing action a in stat s. the term **successor** also used to refer to any state reachable from a given state by a single action. Together, the initial state, actions, and transition model implicitly define the **state space** of the problem. The state space forms a directed network or graph in which the nodes are states and the links between nodes are actions. A **path** in the state space is a sequence of states connected by a sequence of actions.
   4. The **goal test**, which determines whether a given state is a goal state.
   5. A path **cost function** that assigns a numeric cost to each path. A gent chooses a cost function that reflects its own performance measure.
2. **Example Problems:**

   The first example we examine is the vacuum world. This can be formulated as a problem as follows:

   1. **States:** the state is determined by both the agent location and the dirt locations. Agent is in one of two locations, each of which might or might not contain dirt. Thus, there are $2 \times 2^2 = 8$ possible world states.
   2. **Initial state:** any state can be designated as the initial state.

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

3. **Action:** in this simple environment, each state has just three actions: Left, Right, and Suck.
4. **Transition model:** the actions have their expected effects, except that moving left in the leftmost square, moving right in the right most square, and sucking in a clean square have no effect.
5. **Goal test:** this checks whether all squares are clean.
6. **Path cost:** each step cost 1, so the path cost is the number of steps in the path.

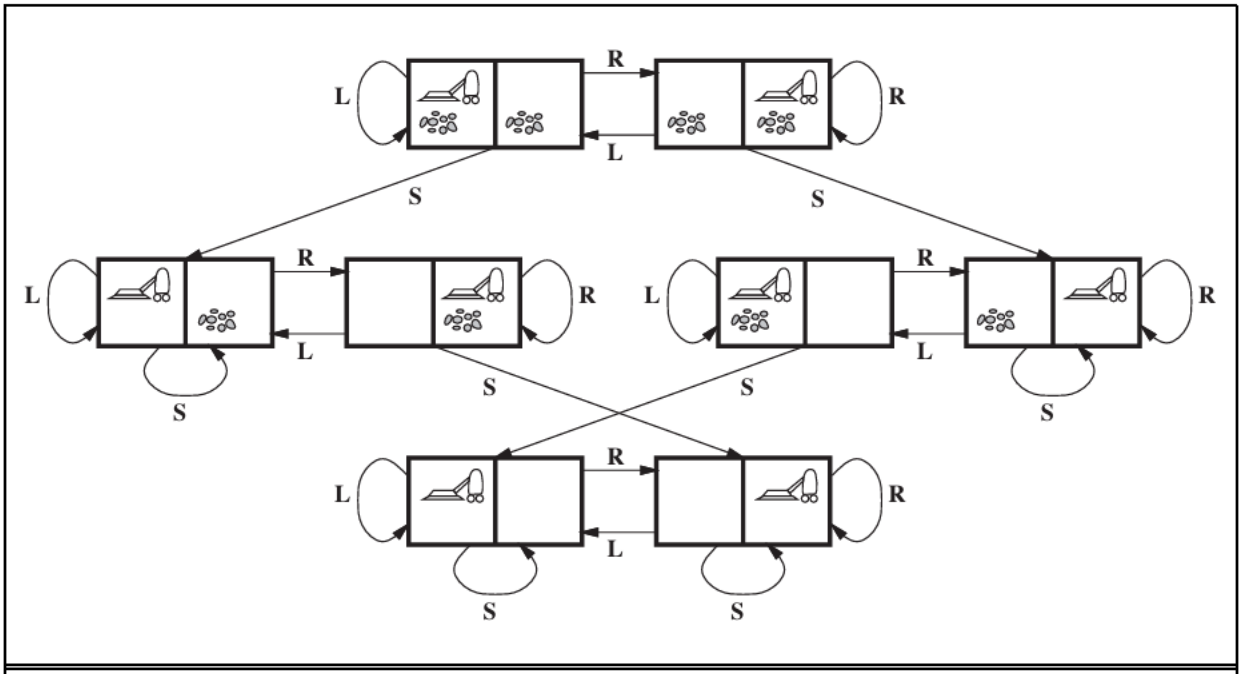

Figure 3.3 The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

Another example is the **8-puzzle**, an instance of which is shown in figure below. It consists of 3 X 3 board with eight numbered tiles and a blank space. A Tile adjacent to the blank space can slide into the space. The object is to reach a specified goal state, such as the one shown on the right of the figure. The standard formulation is as follows:

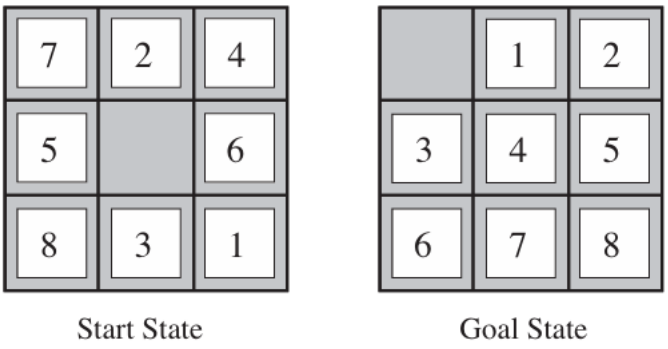

Figure 3.4 A typical instance of the 8-puzzle.

1. **States:** a state description specifies the location of each eight tiles and the blank in one of the nine squares.
2. **Initial state:** any state can be designated as the initial state.
3. **Actions:** the simplest formulation defines the actions as movements of the blank space**: Left, Right, Up, or Down.**
4. **Transition model:** given a state and action, this returns the resulting state, for example if we apply Left to the start state in the above figure, the resulting state has the 5 and the blank switched.
5. **Goal test:** this checks whether the state matches the goal configuration.
6. **Path cost:** each step costs 1, so the path cost is the number of steps in the path.

3. Real-world problems

An example of the real-world problem is the **route-finding** problem. Is defined in terms of specified locations and transitions along links between them. Route-finding algorithms are used in a variety of applications. Some, such as Web sites and in-car system that provide driving directions, routing video streams in computer networks, military operations planning, airline travel-planning systems. Consider the airline travel problem that must be solved by a travel-planning web site.

1. **States:** each state obviously includes a location (airport) and the current time. Furthermore, because the cost of an action may depend on previous action and their status as domestic or international.
2. **Initial state:** this is specified by the user's query.
3. **Actions:** take any flight from the current location, in any seat class, leaving after the current time, leaving enough time for within-airport transfer if needed.
4. **Transition model:** the state resulting from taking a flight will have the flight's destination as the current locations and the flights arrival time as the current time.
5. **Goal test:** are we at the final destination specified by the user.
6. **Path cost:** this depends on cost, waiting time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards and so on.

Other examples of real world may include but not limited to the following:

- Touring problems are closing related to route-finding problems, but with an important difference that is every city must be visited at least once.
- Traveling salesperson problem (TSP) it is a touring problem in which each city must be visited exactly once.
- A VLSI layout problem requires positioning millions of components and connections on a chip to minimize area, minimize circuit delays, minimize stray capacitance, and maximize manufacturing yield.
- Robot navigation it a generalization of the route-finding problem described earlier.
- Automatic assembly sequencing of complex objects by a robot. In assembly problems, the aim is to find an order in which to assemble the parts of some object. If the wrong order is chosen, there will be no way to add some part later in the sequence without undoing some of the work already done.

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 action sequences. The possible action sequences starting at the initial state form a **search tree** with the initial state at the **root**. The branches are **actions** and the **nodes** correspond to states in the state spaced of the problem. Figure below shows the first few steps in growing the search tree for finding a route between two cities.

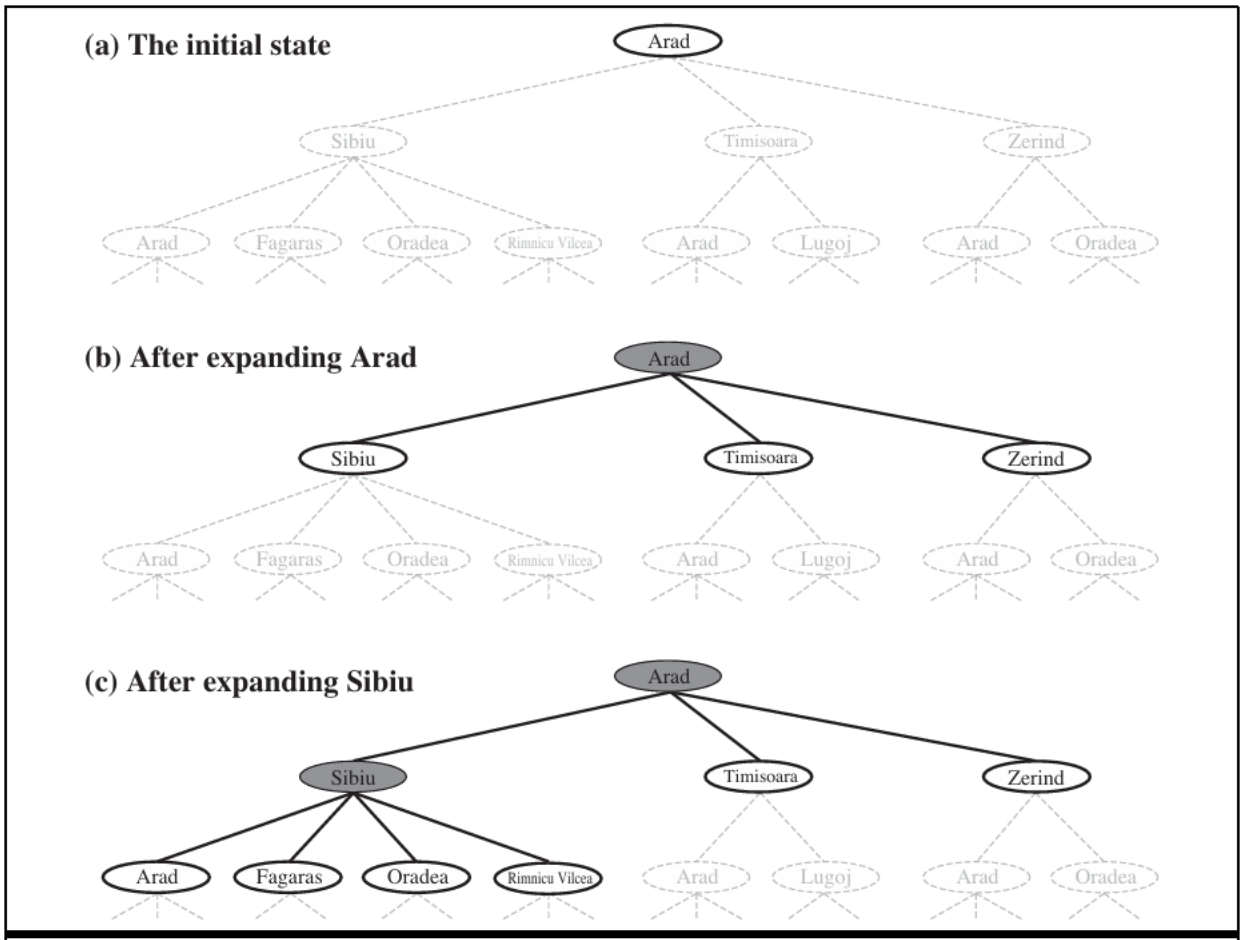

Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

The root node of the tree corresponds to the initial state, In(Arad). The first step is to test whether this is a goal state. Then we take an action by expanding the current

state, that is, applying each legal action to the current state, thereby **generating** a new set of states. In this case, we add three branches from the parent node In(Arad) lead to three new child nodes: In(Sibiu), In(Timisoara), and In(Zerind). Now we must choose which of these three possibilities to consider further.

This is the essence of search algorithm, following up one option now and putting the others aside for later, in case the first choice does not lead to a solution. Suppose we choose Sibiu first. We check to see whether it is a goal state and then expand it to get In(Arad), In(Fagaras), In(Oradea), and In(RimincuVilcea). These nodes are **leaf node**, that is , a node with no children in the tree. The set of all nodes available for expansion is called **frontier**. The process of expanding nodes on the frontier continues until either a solution is found or there are no more states to expands.

3. **Measuring problem-solving performance:** before we get into the design of specific search algorithms, we need to consider the criteria that might be used to choose among them, we can evaluate an algorithms performance in four ways:
   1. **Completeness: is the algorithm guaranteed to find a solution when there is one.**
   2. **Optimality: does the strategy find the optimal solution.**
   3. **Time complexity: how long does it take to find a solution**
   4. **Space complexity: how much memory is needed to perform the search.**

UNINFORMED SEARCH STRATEGIES (Blind Search)

This section covers several search strategies that come under the heading of uninformed search (also called blind search). The term means that the strategies have no additional information about states beyond that provided in the problem definition. All they can do is generate successors and distinguish a goal state from a non-goal state. ***All search strategies are distinguished by the order in which nodes are expanded***. Strategies that know whether one-goal state is “more promising” than another are called **informed search or heuristic** search strategies, they are covered in the next lecture.

1. **Breadth-first search:** it is a simple strategy in which the root node is expanded first, then all the successors of the root 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 algorithm the *shallowest* unexpanded node is chosen for expansion. This is achieved very simply by using a FIFO queue for the frontier nodes. Thus, new nodes which are always deeper than their parents go to the back of the queue, and old nodes, which are shallower than the new nodes get expanded first.

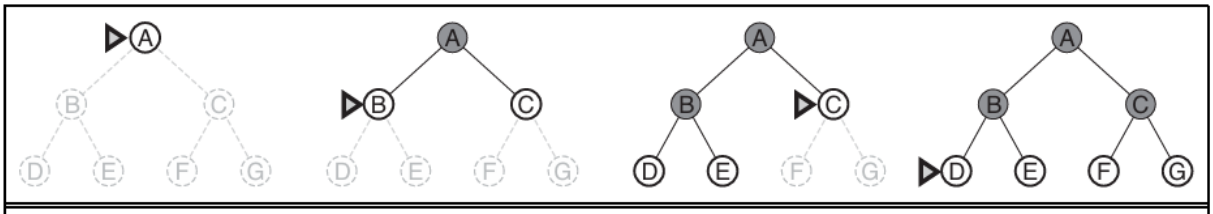

Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

How does breadth-first search rate according to the four criteria from the previous section? We can easily see that it is complete, breadth-first search will eventually find the goal node after generating all shallower nodes. Now, the shallowest goal node is not necessarily the optimal one, technically, breadth-first search is optimal if the path cost is a nondecreasing function of the depth of the node. The most common such scenario is that all actions have the same cost. On the other hand, time and memory, criteria are not good. Figure below shows why.

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

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

2. **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 storin the frontier nodes 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. The first is that the goal test is applied to a node when it is selected for expansion. The second difference is that a test is added in case a better path found to a node currently on the frontier. the algorithm is shown in figure below.

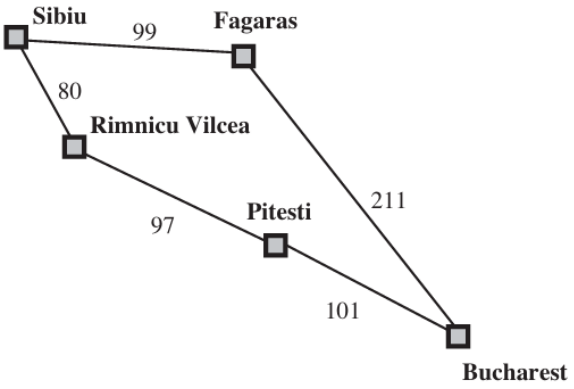

Figure 3.15 Part of the Romania state space, selected to illustrate uniform-cost search.

The above figure explains the problem to get from Sibiu to Bucharest. The successors of Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 90, respectively. the least cost node Rimnicu Vilcea is expanded next, adding Pitesti with cost 80 + 97 = 177. The least cost 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 expansion and solution is returned.

3. **Depth-first search:** it always expands the deepest node in the current frontier of the search tree. The progress of the search is illustrated in figure below.

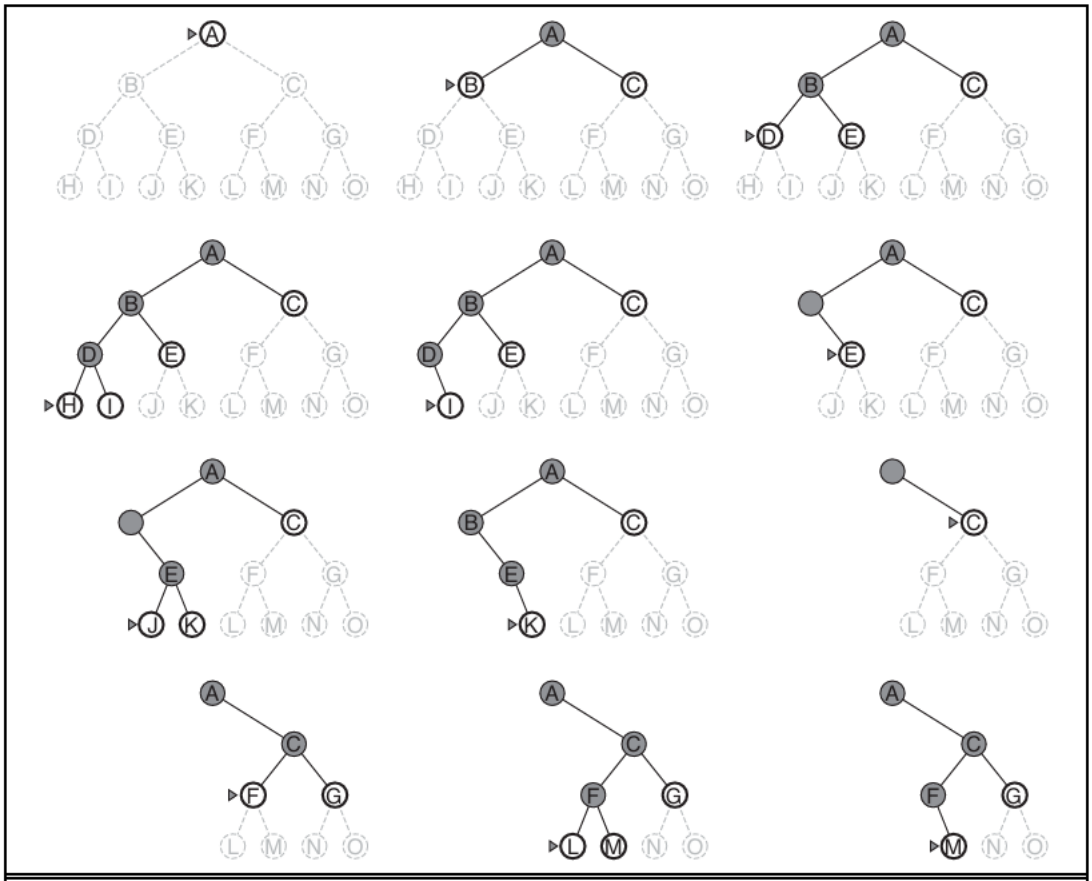

Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

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 frontier, so then the search “backs up” to the next deepest node that still has unexplored successors.

Whereas breadth-first-search uses a FIFO queue, 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 wat the deepest unexpanded node when it was selected.

It can be seen that depth-first search will explore the entire left subtree even if node C is a goal node. If node J where also a goal node then depth-first search would return it as a solution instead of C, which would be a better solution, hence depth first search is not optimal. Additionally, depth-first search may generate all of the nodes in the search tree, where m is the maximum depth of any node. This can be much greater than the size of the state space.

4. **Depth-limited search:** to overcome the drawbacks of depth-first search in infinite state space can be alleviated by supplying depth-first search with a predetermined depth limit *L*, that is, nodes at depth l are treated as if they have not successors. This approach is called **depth-limited search**. Even though, the depth limit solves some problem, it also introduces an additional source of incompleteness. If we choose *L* < d, that is the shallowest goal is beyond the depth limit. Depth-limit search will also be nonoptimal if we choose *L* > d. 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 it there is a solution, it must be of length 19 at the longest possible choice. So *L = 19.*
5. **Iterative deepening depth-first search:** it is a general strategy, often used in combination with depth-first tree search, that finds the bets depth limit. Id 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. Iterative deepening combines the benefits of depth-first and breadth-first search. Like depth-first search, its memory requirements are modest. Like breadth-first search, it is complete and optimal when the path cost is a nondecreasing function of the depth of the node.

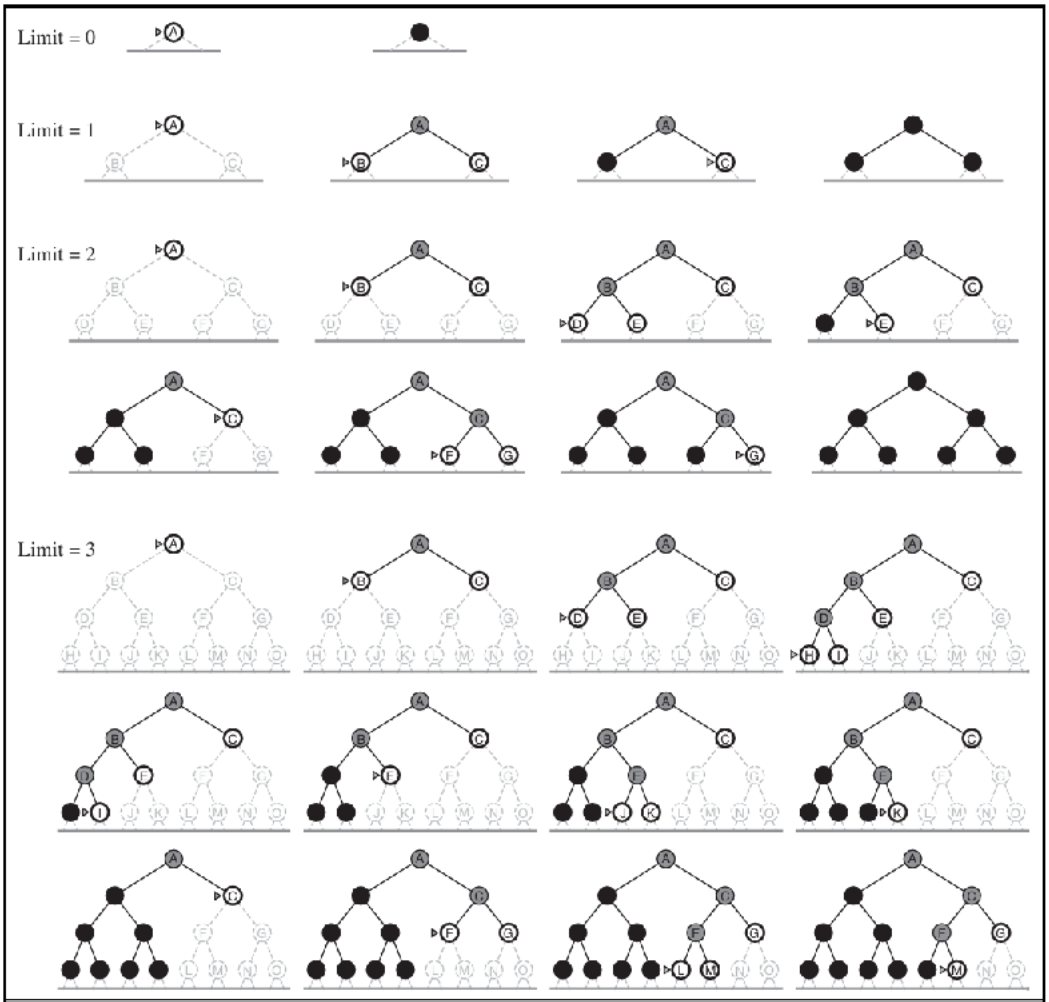

Figure 3.19 Four iterations of iterative deepening search on a binary tree.

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.

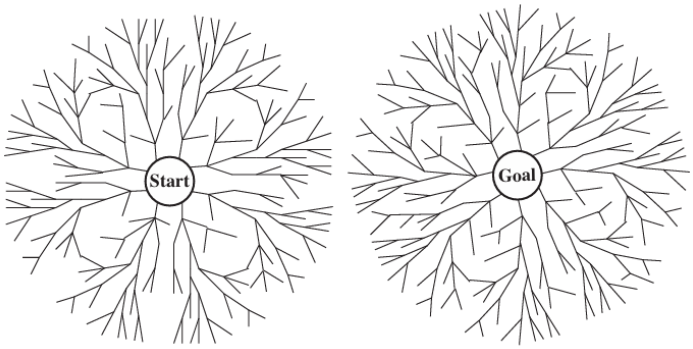

Figure 3.20 A schematic view of a bidirectional search that is about to succeed when a branch from the start node meets a branch from the goal node.

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.