Problem Solving By Intelligent Search

*Problem solving requires two prime considerations: first representation of the problem by an appropriately organized state space and then testing the existence of a well-defined goal state in that space. Identification of the goal state and determination of the optimal path, leading to the goal through one or more transitions from a given starting state, will be addressed in this chapter in sufficient details. The chapter, thus, starts with some well-known search algorithms, such as the depth first and the breadth first search, with special emphasis on their results of time and space complexity. It then gradually explores the '*heuristic search*' algorithms, where the order of visiting the states in a search space is supported by thumb rules, called heuristics, and demonstrates their applications in complex problem solving. It also discusses some intelligent search algorithms for game playing.*

Introduction

We have already come across some of the problems that can be solved by intelligent search. For instance, the well-known water-jug problem, the number puzzle problem and the missionaries-cannibals problem are ideal examples of problems that can be solved by intelligent search. Common experience reveals that a search problem is associated with two important issues: first 'what to search' and secondly 'where to search'. The first one is generally referred to as 'the key', while the second one is termed 'search space'. In AI the search space is generally referred to as a collection of states and is thus called state space. Unlike common search space, the state space in
most of the problems in AI is not completely known, prior to solving the problem. So, solving a problem in AI calls for two phases: the generation of the space of states and the searching of the desired problem state in that space. Further, since the whole state space for a problem is quite large, generation of the whole space prior to search may cause a significant blockage of storage, leaving a little for the search part. To overcome this problem, the state space is expanded in steps and the desired state, called "the goal", is searched after each incremental expansion of the state space.

To successfully design and implement search algorithms, a programmer must be able
to analyze and predict the behavior of the problem. Questions that need to be answered include:

- Is the problem solver guaranteed to find a solution?
- Will the problem solver always terminate? Can it become caught in an infinite loop?
- When a solution is found, is it guaranteed to be optimal?
- What is the complexity of the search process in terms of time usage? Memory usage?

- How can the interpreter most effectively reduce search complexity?
- How can an interpreter be designed to most effectively utilize a representation Language?

The theory of *state space search* is our primary tool for answering these questions. By representing a problem as a *state space graph*, we can use *graph theory* to analyze the structure and complexity of both the problem and the search procedures that we employ to solve it.

A graph consists of a set of *nodes* and a set of *arcs* or *links* connecting pairs of nodes. In the state space model of problem solving, the nodes of a graph are taken to represent discrete *states* in a problem-solving process, such as the results of logical inferences or the different configurations of a game board. The arcs of the graph represent transitions between states. These transitions correspond to logical inferences or legal moves of a game. In expert systems, for example, states describe our knowledge of a problem instance at some stage of a reasoning process. Expert knowledge, in the form of ***if . . . then*** rules, allows us to generate new information; the act of applying a rule is represented as an arc between states.

Graph theory is our best tool for reasoning about the structure of objects and relations; indeed, this is precisely the need that led to its creation in the early eighteenth century. The Swiss mathematician Leonhard Euler invented graph theory to solve the "bridges of Konigsberg problem." The city of Konigsberg occupied both banks and two islands of a river. The islands and the riverbanks were connected by seven bridges, as indicated in Figure (5-1).

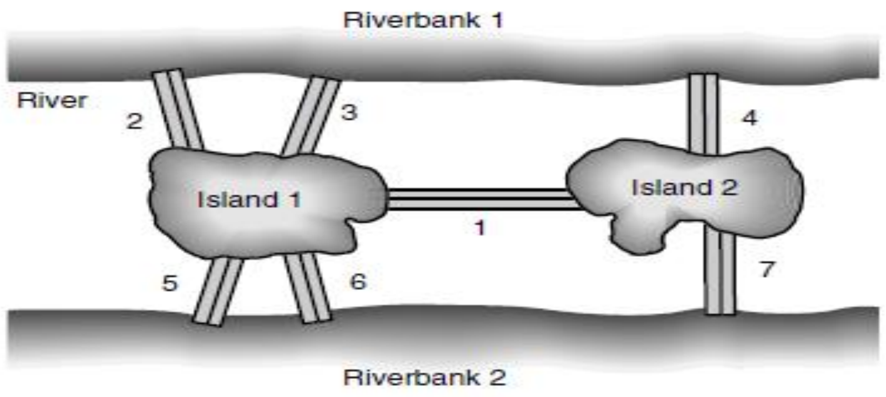


Figure (5-1): The city of Konigsberg

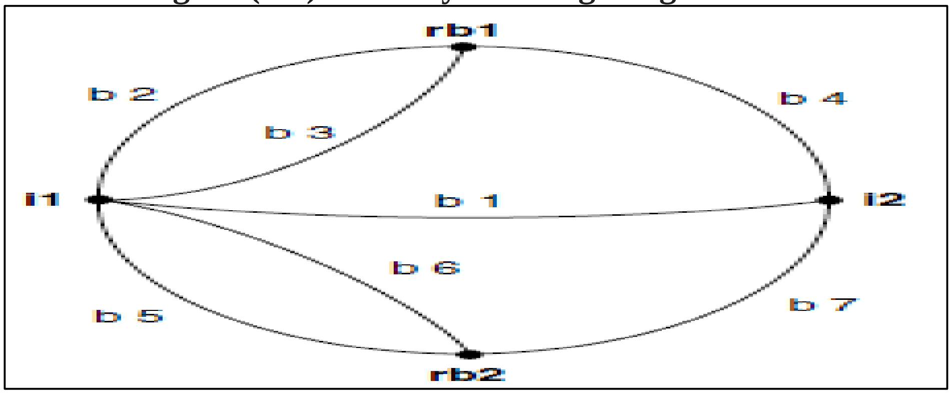


Figure (5-2): Graph of the Konigsberg bridge system.

The bridges of Konigsberg problem asks if there is a walk around the city that crosses each bridge exactly once. Although the residents had failed to find such a walk and doubted that it was possible, no one had proved its impossibility. Devising a form of graph theory, Euler created an alternative representation for the map, presented in Figure 6.2. The riverbanks (rb1 and rb2) and islands (i1 and i2) are described by the nodes of a graph; the bridges are represented by labeled arcs between nodes (b1, b2, , b7). The graph representation preserves the essential structure of the bridge system, while ignoring extraneous features such as bridge lengths, distances, and order of bridges in the walk.

Alternatively, we may represent the Konigsberg bridge system using predicate calculus. The connect predicate corresponds to an arc of the graph, asserting that two land masses are connected by a particular bridge. Each bridge requires two connect predicates, one for each direction in which the bridge may be crossed. A predicate expression, connect(X, Y, Z) = connect (Y, X, Z), indicating that any bridge can be crossed in either direction, would allow removal of half the following connect facts:

connect(i1, i2, b1)	connect(i2, i1, b1)
connect(rb1, i1, b2)	connect(i1, rb1, b2)
connect(rb1, i1, b3)	connect(i1, rb1, b3)
connect(rb1, i2, b4)	connect(i2, rb1, b4)
connect(rb2, i1, b5)	connect(i1, rb2, b5)
connect(rb2, i1, b6)	connect(i1, rb2, b6)
connect(rb2, i2, b7)	connect(i2, rb2, b7)

The predicate calculus representation is equivalent to the graph representation in that the connectedness is preserved. Indeed, an algorithm could translate between the two representations with no loss of information. However, the structure of the problem can be visualized more directly in the graph representation, whereas it is left implicit in the predicate calculus version. Euler’s proof illustrates this distinction.

In proving that the walk was impossible, Euler focused on the *degree* of the nodes of the graph, observing that a node could be of either *even* or *odd* degree. An *even* degree node has an even number of arcs joining it to neighboring nodes. An *odd* degree node has an odd number of arcs. With the exception of its beginning and ending nodes, the desired walk would have to leave each node exactly as often as it entered it. Nodes of odd degree could be used only as the beginning or ending of the walk, because such nodes could be crossed only a certain number of times before they proved to be a dead end. The traveler could not exit the node without using a previously traveled arc.

Euler noted that unless a graph contained either exactly zero or two nodes of odd degree, the walk was impossible. If there were two odd-degree nodes, the walk could start at the first and end at the second; if there were no nodes of odd degree, the walk could begin and end at the same node. The walk is not possible for graphs containing any other number of nodes of odd degree, as is the case with the city of Konigsberg. This problem is now called finding an Euler path through a graph.
Note that the predicate calculus representation, though it captures the relationships between bridges and land in the city, does not suggest the concept of the degree of a node. In the graph representation there is a single instance of each node with arcs between the nodes, rather than multiple occurrences of constants as arguments in a set of predicates. For this reason, the graph representation suggests the concept of node degree and the focus of Euler's proof. This illustrates graph theory's power for analyzing the structure of objects, properties, and relationships.

Structure for the state space search:

There are many ways to describe state space and the most common one is the graph to clear that topic we need to go through graph theory:

Graph Theory

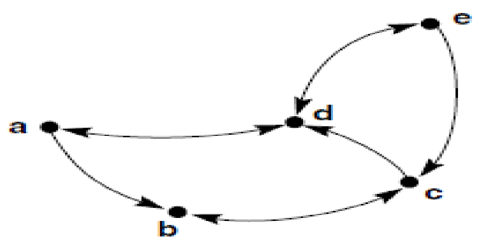
A *graph* is a set of *nodes* or *states* and a set of arcs that connect the nodes. A *labeled graph* has one or more descriptors (labels) attached to each node that distinguishes that node from any other node in the graph. In a *state space graph*, these descriptors identify states in a problem-solving process. If there are no descriptive differences between two nodes, they are considered the same. The arc between two nodes is indicated by the labels of the connected nodes.

The arcs of a graph may also be labeled. Arc labels are used to indicate that an arc represents a named relationship (as in a semantic network) or to attach weights to arcs (as in the traveling salesperson problem). If there are different arcs between the same two nodes (as in Figure (5-2)), these can also be distinguished through labeling.

A graph is ***directed*** if arcs have an associated directionality. The arcs in a directed graph are usually drawn as arrows or have an arrow attached to indicate direction. Arcs that can be crossed in either direction may have two arrows attached but more often have no direction indicators at all. Figure (5-3) is a labeled, directed graph: arc **(a, b)** may only be crossed from node **a** to node **b**, but arc **(b, c)** is crossable in either direction.
A ***path*** through a graph connects a sequence of nodes through successive arcs. The
Path is represented by an ordered list that records the nodes in the order they occur in the path. In Figure (5-3) **[a, b, c, d]** represents the path through nodes a, b, c, and d, in that order.

A *rooted* graph has a unique node, called the *root*, such that there is a path from the root to all nodes within the graph. In drawing a rooted graph, the root is usually drawn at the top of the page, above the other nodes. The state space graphs for games are usually rooted graphs with the start of the game as the root.

A *tree* is a graph in which two nodes have at most one path between them. Trees often have roots, in which case they are usually drawn with the root at the top, like a rooted graph. Because each node in a tree has only one path of access from any other node, it is impossible for a path to *loop* or *cycle* through a sequence of nodes.



Nodes = **{a,b,c,d,e}**
Arcs = **{(a,b),(a,d),(b,c),(c,b),(c,d),(d,a),(d,e),(e,c),(e,d)}**

Figure (5.3) A labeled directed graph.

For rooted trees or graphs, relationships between nodes include parent, *child*, and *sibling*. These are used in the usual familial fashion with the parent preceding its child along a directed arc. The children of a node are called *siblings*. Similarly, an *ancestor* comes before a *descendant* in some path of a directed graph. In Figure (5-4), **b** is a parent of nodes **e** and **f** (which are, therefore, children of **b** and siblings of each other). Nodes **a** and **c** are ancestors of states **g, h**, and **i**, and **g, h,** and **i** are descendants of a and **c**.

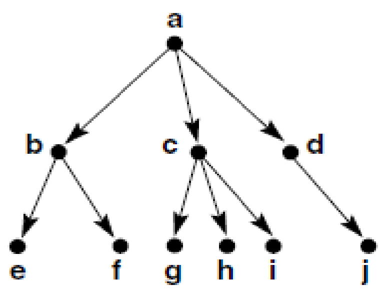


Figure (5.4) A rooted tree, exemplifying Family relationships.

Before introducing the state space representation of problems we formally define these concepts.

D E F I N I T I O N

GRAPH

A graph consists of:

A set of *nodes* N1, N2, N3, ..., Nn, ..., which need not be finite.

A set of *arcs* that connect pairs of nodes.

Arcs are ordered pairs of nodes; i.e., the arc (N3, N4) connects node N3 to node N4. This indicates a direct connection from node N3 to N4 but not from N4 to N3, unless (N4, N3) is also an arc, and then the arc joining N3 and N4 is *undirected*.

If a *directed* arc connects N_j and N_k, then N_j is called the parent of N_k and N_k, the child of N_j. If the graph also contains an arc (N_j, N_l), then N_k and N_l are called *siblings*.

A rooted graph has a unique node N_s from which all paths in the graph originate. That is, the root has no parent in the graph.

A *tip* or *leaf* node is a node that has no children.

An ordered sequence of nodes [N_1, N_2, N_3, ..., N_n], where each pair N_i, N_{i+1} in the sequence represents an arc, i.e., (N_i, N_{i+1}), is called a *path* of length $n - 1$.

On a path in a rooted graph, a node is said to be an *ancestor* of all nodes positioned after it (to its right) as well as a *descendant* of all nodes before it.

A path that contains any node more than once (some N_j in the definition of path above is repeated) is said to contain a *cycle* or *loop*.

A *tree* is a graph in which there is a unique path between every pair of nodes. (The paths in a tree, therefore, contain no cycles.)

The edges in a rooted tree are directed away from the root. Each node in a rooted tree has a unique parent.

Two nodes are said to be connected if a path exists that includes them both.

The State Space Representation of Problems

In the *state space representation* of a problem, the nodes of a graph correspond to partial problem solution *states* and the arcs correspond to steps in a problem-solving process. One or more *initial states*, corresponding to the given information in a problem instance, form the root of the graph. The graph also defines one or more *goal* conditions, which are solutions to a problem instance. *State space search* characterizes problem solving as the process of finding a *solution path* from the start state to a goal.

We now formally define the state space representation of problems:
D E F I N I T I O N
STATE SPACE SEARCH
A *state space* is represented by a four-tuple **[N,A,S,GD]**, where:

N is the set of nodes or states of the graph. These correspond to the states in a problem-solving process.
A is the set of arcs (or links) between nodes. These correspond to the steps in a problem-solving process.
S, a nonempty subset of **N**, contains the start state(s) of the problem.

GD, a nonempty subset of **N**, contains the goal state(s) of the problem. The states in GD are described using either:

1. A measurable property of the states encountered in the search.
2. A measurable property of the path developed in the search, for example, the sum of the transition costs for the arcs of the path.

A *solution path* is a path through this graph from a node in S to a node in GD.

A goal may describe a state, such as a winning board in tic-tac-toe (Figure (5-5)) or a goal configuration in the 8-puzzle (Figure (5-6)). Alternatively, a goal can describe some property of the solution path itself.

Arcs of the state space correspond to steps in a solution process and paths through the space represent solutions in various stages of completion. Paths are searched, beginning at the start state and continuing through the graph, until either the goal description is satisfied or they are abandoned. The actual generation of new states along the path is done by applying operators, such as ″ legal moves″ in a game or inference rules in a logic problem or expert system, to existing states on a path.

The task of a search algorithm is to find a solution path through such a problem space. Search algorithms must keep track of the paths from a start to a goal node, because these paths contain the series of operations that lead to the problem solution.

One of the general features of a graph, and one of the problems that arise in the design of a graph search algorithm, is that states can sometimes be reached through different paths. For example, in Figure (5-3) a path can be made from state **a** to state **d** either through **b** and **c** or directly from **a** to **d**. This makes it important to choose the *best path* according to the needs of a problem. In addition, multiple paths to a state can lead to loops or cycles in a solution path that prevent the algorithm from reaching a goal. A blind search for goal state e

in the graph of Figure (5-3) might search the sequence of states **abcdabcdabcd . . .** forever!If the space to be searched is a tree, as in Figure (5-4), the problem of cycles does not occur. It is, therefore, important to distinguish between problems whose state space is a tree and those that may contain loops. General graph search algorithms must detect and eliminate loops from potential solution paths, whereas tree searches may gain efficiency by eliminating this test and its overhead.
Tic-tac-toe and the *8-puzzle* exemplify the state spaces of simple games. Both of these examples demonstrate termination conditions of *type 1* in our definition of state space search. Example 5.1.3, the traveling salesperson problem, has a goal description of *type 2*, the total cost of the path itself.

EXAMPLE 5.1.1: TIC-TAC-TOE
The state space representation of tic-tac-toe appears in Figure (5-5) The start state is an empty board, and the termination or goal description is a board state having three **Xs** in a row, column, or diagonal (assuming that the goal is a win for X). The path from the start state to a goal state gives the series of moves in a winning game.
The states in the space are all the different configurations of Xs and Os that the game can have. Of course, although there are 3^9 *ways to arrange {blank, X, O} in nine spaces, most of them would never occur in an actual game. Arcs are generated by legal moves of the game, alternating between placing an X and an O in an unused location. The state space is a graph rather than a tree, as some states on the third and deeper levels can be reached by different paths. However, there are no cycles in the state space, because the directed arcs of the graph do not allow a move to be undone. It is impossible to "go back up" the structure once a state has been reached. No checking for cycles in path generation is necessary.*

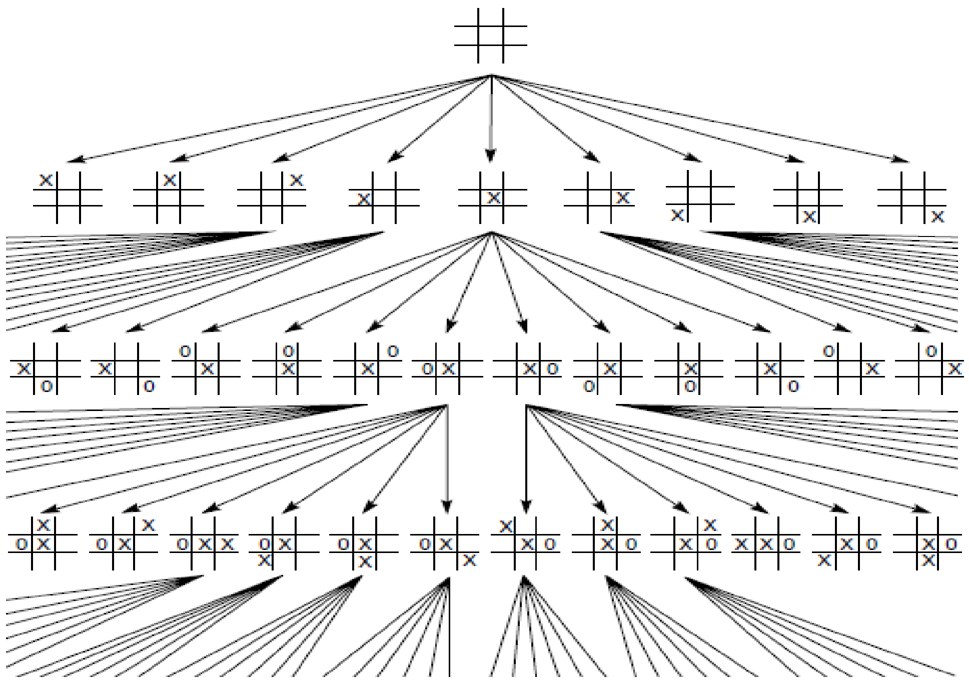


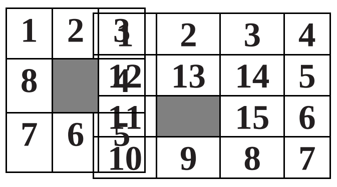
Figure (5-5) **Portion of the state space for tic-tac-toe.**

The state space representation provides a means of determining the complexity of the problem. In tic-tac-toe, there are nine first moves with eight possible responses to each of them, followed by seven possible responses to each of these, and so on. It follows that 9 × 8 × 7 × ... or 9! different paths can be generated. Although it is not impossible for a computer to search this number of paths (362,880) exhaustively, many important problems also exhibit factorial or exponential complexity, although on a much larger scale. Chess has 10^{120} possible game paths; checkers has10^{40}, some of which may never occur in an actual game. These spaces are difficult or impossible to search exhaustively. Strategies for searching such large spaces often rely on heuristics to reduce the complexity of the search.

EXAMPLE 5.1.2: THE 8-PUZZLE

In the *15-puzzle* of Figure (5-6), 15 differently numbered tiles are fitted into 16 spaces on a grid. One space is left blank so that tiles can be moved around to form different patterns. The goal is to find a series of moves of tiles into the blank space that places the board in a goal configuration. This is a common game that most of us played as children. (The version I remember was about 3 inches square and had red and white tiles in a black frame.)

A number of interesting aspects of this game have made it useful to researchers in problem solving. The state space is large enough to be interesting but is not completely intractable (16! if symmetric states are treated as distinct). Game states are easy to represent.



15-puzzle **8-puzzle**

Figure (5-6) the 15-puzzle and the 8-puzzle.

The 8-puzzle is a 3 × 3 version of the 15-puzzle in which eight tiles can be moved around in nine spaces. Because the 8-puzzle generates a smaller state space than the full 15-puzzle and its graph fits easily on a page, it is used for many examples in this lecture notes.

Although in the physical puzzle moves are made by moving tiles ("move the 7 tile right, provided the blank is to the right of the tile" or "move the 3 tile down”), it is much simpler to think in terms of “moving the blank space”. This simplifies the definition of move rules because there are eight tiles but only a single blank. In order to apply a move, we must make sure that it does not move the blank off the board. Therefore, all four moves are not applicable at all times; for example, when the blank is in one of the corners only two moves are possible.

The legal moves are:

- **move the blank up** ↑
- **move the blank right** →
- **move the blank down** ↓
- **move the blank left** ←

If we specify a beginning state and a goal state for the 8-puzzle, it is possible to give a state space accounting of the problem-solving process (Figure 5-7). States could be:

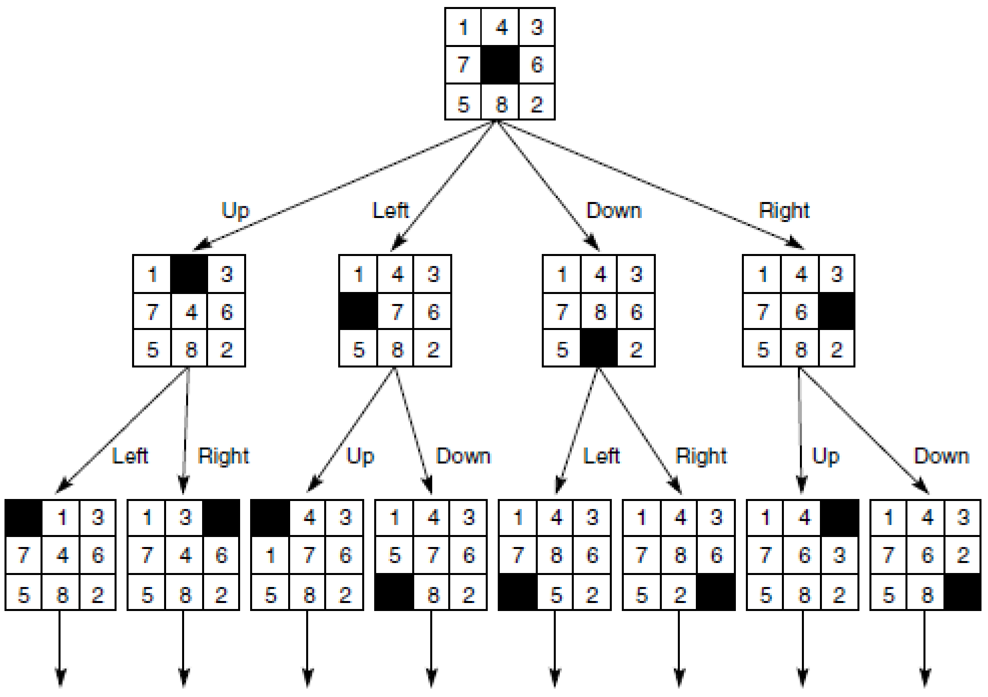


Figure (5-7) State space of the 8-puzzle generated by move blank operations.

Represented using a simple 3 × 3 array. A predicate calculus representation could use a “state” predicate with nine parameters (for the locations of numbers in the grid). Four procedures, describing each of the possible moves of the blank, define the arcs in the state space.

As with tic-tac-toe, the state space for the 8-puzzle is a graph (with most states having multiple parents), but unlike tic-tac-toe, cycles are possible. The GD or goal description of the state space is a particular state or board configuration. When this state is found on a path, the search terminates. The path from start to goal is the desired series of moves.

It is interesting to note that the complete state space of the 8- and 15-puzzles consists of two disconnected (and in this case equal-sized) subgraphs. This makes half the possible states in the search space impossible to reach from any given start state. If we exchange (by prying loose!) two immediately adjacent tiles, states in the other component of the space become reachable.

EXAMPLE 5.1.3: THE TRAVELING SALESPERSON

Suppose a salesperson has five cities to visit and then must return home. The goal of the problem is to find the shortest path for the salesperson to travel, visiting each city, and then returning to the starting city. Figure (5-8) gives an instance of this problem. The nodes of the graph represent cities, and each arc is labeled with a weight indicating the cost of traveling that arc. This cost might be a representation of the miles necessary in car travel or cost of an air flight between the two cities. For convenience, we assume the salesperson lives in city A and will return there, although this assumption simply reduces the problem of N cities to a problem of (N - 1) cities.

The path [A,D,C,B,E,A], with associated cost of 450 miles, is an example of a possible circuit. The goal description requires a complete circuit with minimum cost. Note

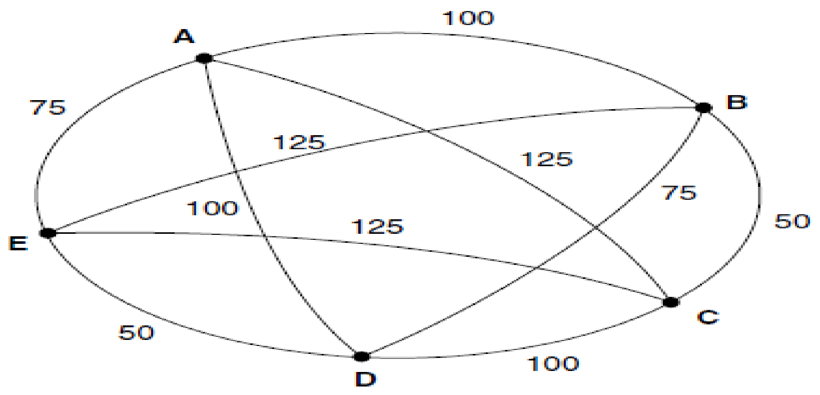
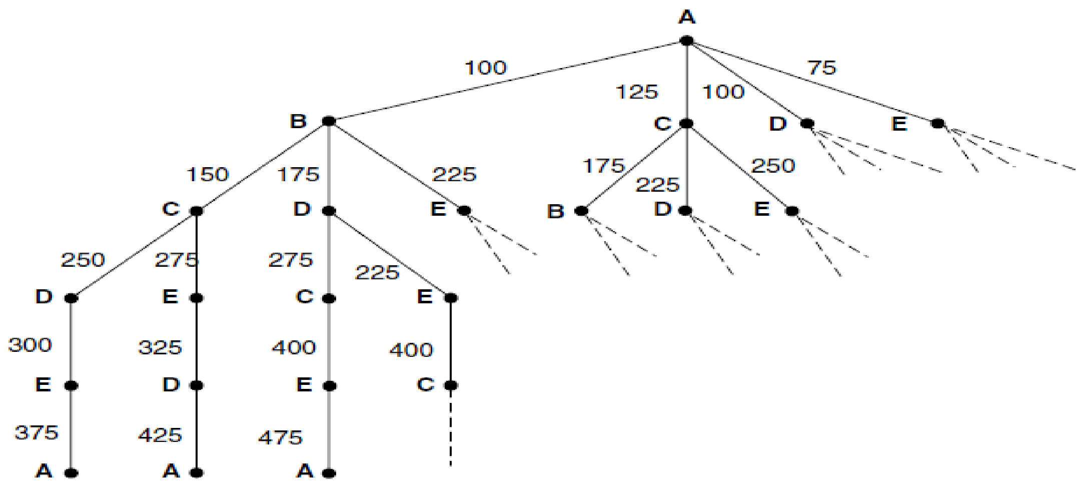


Figure (5-8) an instance of the traveling salesperson problem.



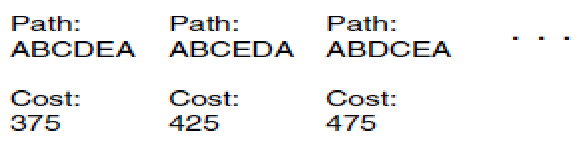


Figure (5-9) Search of the traveling salesperson problem. Each arc is marked with the total weight of all paths from the start node (A) to its endpoint.

that the goal description is a property of the entire path, rather than of a single state. This is a goal description of type 2 from the definition of state space search.

Figure (5-9) shows one way in which possible solution paths may be generated and compared. Beginning with node A, possible next states are added until all cities are included and the path returns home. The goal is the lowest-cost path.

As Figure (5-9) suggests, the complexity of exhaustive search in the traveling salesperson problem is (N - 1)!, where N is the number of cities in the graph. For 9 cities we may exhaustively try all paths, but for any problem instance of interesting size, for example with 50 cities, simple exhaustive search cannot be performed within a practical length of time. In fact complexity costs for an N! search grow so fast that very soon the search combinations become intractable.

Several techniques can reduce this search complexity. One is called branch and bound (Horowitz and Sahni 1978). Branch and bound generates paths one at a time, keeping track of the best circuit found so far. This value is used as a bound on future candidates. As paths are constructed one city at a time, the algorithm examines each partially completed path. If the algorithm determines that the best possible extension to a path, the branch, will have greater cost than the bound, it eliminates that partial path and all of its possible extensions. This reduces search considerably but still leaves an exponential number of paths (1.26^N rather than N!).

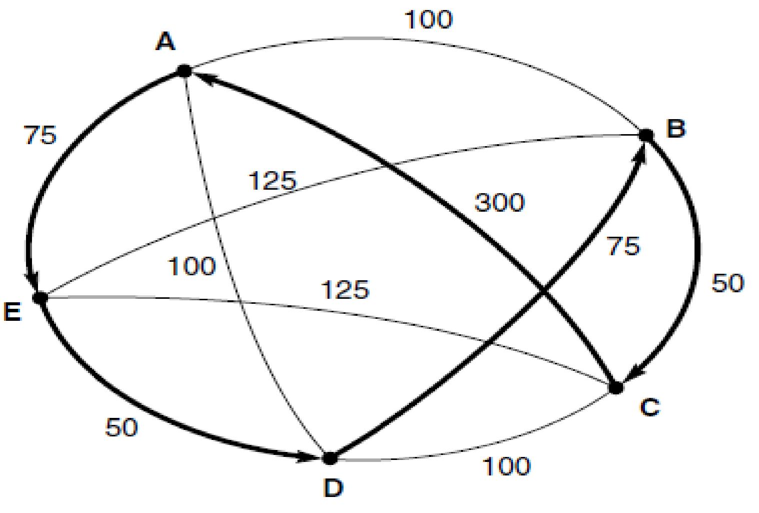


Figure (5-10) An instance of the traveling salesperson problem with the nearest neighbor path in bold. Note that

this path (A, E, D, B, C, A), at a cost of 550, is not
the shortest path. The comparatively high cost of
arc (C, A) defeated the heuristic.

Another strategy for controlling search constructs the path according to the rule "go to the closest unvisited city." The *nearest neighbor* path through the graph of Figure 5-10 is **[A,E,D,B,C,A],** at a cost of 375 miles. This method is highly efficient, as there is only one path to be tried! The nearest neighbor, sometimes called *greedy*, heuristic is fallible, as graphs exist for which it does not find the shortest path, see Figure 3.11, but it is a possible compromise when the time required makes exhaustive search impractical.

5.2 Strategies for State Space Search

5.2.1 Data-Driven and Goal-Driven Search

A state space may be searched in two directions: from the given data of a problem instance toward a goal or from a goal back to the data.

In *data-driven search*, sometimes called *forward chaining*, the problem solver begins with the given facts of the problem and a set of legal moves or rules for changing state. Search proceeds by applying rules to facts to produce new facts, which are in turn used by the rules to generate more new facts. This process continues until (we hope!) it generates a path that satisfies the goal condition.

An alternative approach is possible: take the goal that we want to solve. See what rules or legal moves could be used to generate this goal and determine what conditions must be true to use them. These conditions become the new goals, or subgoals, for the search. Search continues, working backward through successive subgoals until (we hope!) it works back to the facts of the problem. This finds the chain of moves or rules leading from data to a goal, although it does so in backward order. This approach is called goal-driven reasoning, or backward chaining, and it recalls the simple childhood trick of trying to solve a maze by working back from the finish to the start.

To summarize: data-driven reasoning takes the facts of the problem and applies the rules or legal moves to produce new facts that lead to a goal; goal-driven reasoning focuses on the goal, finds the rules that could produce the goal, and chains backward through successive rules and subgoals to the given facts of the problem.

In the final analysis, both data-driven and goal-driven problem solvers search the same state space graph; however, the order and actual number of states searched can differ. The preferred strategy is determined by the properties of the problem itself. These include the complexity of the rules, the "shape" of the state space, and the nature and availability of the problem data. All of these vary for different problems.

As an example of the effect a search strategy can have on the complexity of search, consider the problem of confirming or denying the statement ″I am a descendant of Thomas Jefferson.″ A solution is a path of direct lineage between the "I" and Thomas Jefferson. This space may be searched in two directions, starting with the "I" and working along ancestor lines to Thomas Jefferson or starting with Thomas Jefferson and working through his descendants.

Some simple assumptions let us estimate the size of the space searched in each direction. Thomas Jefferson was born about 250 years ago; if we assume 25 years per generation, the required path will be about length 10. As each person has exactly two parents, a search back from the "I" would examine on the order of 2^{10} ancestors. A search that worked forward from Thomas Jefferson would examine more states, as people tend to have more than two children (particularly in the eighteenth and nineteenth centuries). If we assume an average of only three children per family, the search would examine on the order of 3^{10} nodes of the family tree. Thus, a search back from the "I" would examine fewer nodes. Note, however, that both directions yield exponential complexity.

The decision to choose between data- and goal-driven search is based on the structure of the problem to be solved. Goal-driven search is suggested if:

1. A goal or hypothesis is given in the problem statement or can easily be formulated. In a mathematics theorem prover, for example, the goal is the theorem to be proved. Many diagnostic systems consider potential diagnoses in a systematic fashion, confirming or eliminating them using goal-driven reasoning.

2. There are a large number of rules that match the facts of the problem and thus produce an increasing number of conclusions or goals. Early selection of a goal can eliminate most of these branches, making goal-driven search more effective in pruning the space (Figure 5-11). In a theorem prover, for example, the total number of rules used to produce a given theorem is usually much smaller than the number of rules that may be applied to the entire set of axioms.

3. Problem data are not given but must be acquired by the problem solver. In this case, goal-driven search can help guide data acquisition. In a medical diagnosis program, for example, a wide range of diagnostic tests can be applied. Doctors order only those that are necessary to confirm or deny a particular hypothesis.

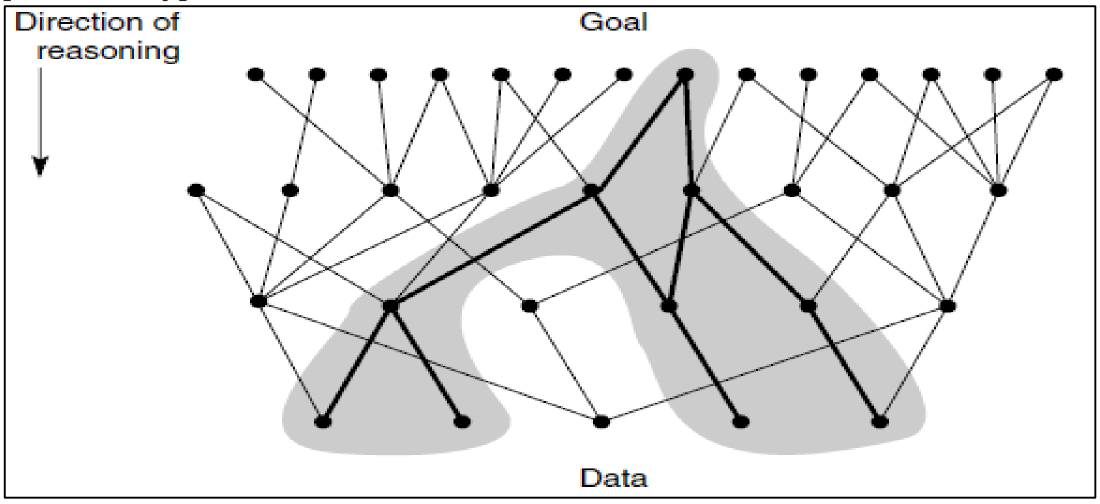


Figure (5-11) State space in which goal-directed search effectively prunes extraneous search paths.

Goal-driven search thus uses knowledge of the desired goal to guide the search through relevant rules and eliminate branches of the space.

Data-driven search (Figure 5-12) is appropriate for problems in which:

1. All or most of the data are given in the initial problem statement. Interpretation problems often fit this mold by presenting a collection of data and asking the system to provide a high-level interpretation. Systems that analyze particular data (e.g., the PROSPECTOR or Dipmeter programs, which interpret geological data or attempt to find what minerals are likely to be found at a site) fit the data-driven approach.
2. There are a large number of potential goals, but there are only a few ways to use the facts and given information of a particular problem instance. The DENDRAL program, an expert system that finds the molecular structure of organic compounds based on their formula, mass spectrographic data, and knowledge of chemistry, is an example of this. For any organic compound, there are an enormous number of possible structures. However, the mass spectrographic data on a compound allow DENDRAL to eliminate all but a few of these.
3. It is difficult to form a goal or hypothesis. In using DENDRAL, for example, little may be known initially about the possible structure of a compound.

Data-driven search uses the knowledge and constraints found in the given data of a problem to guide search along lines known to be true.

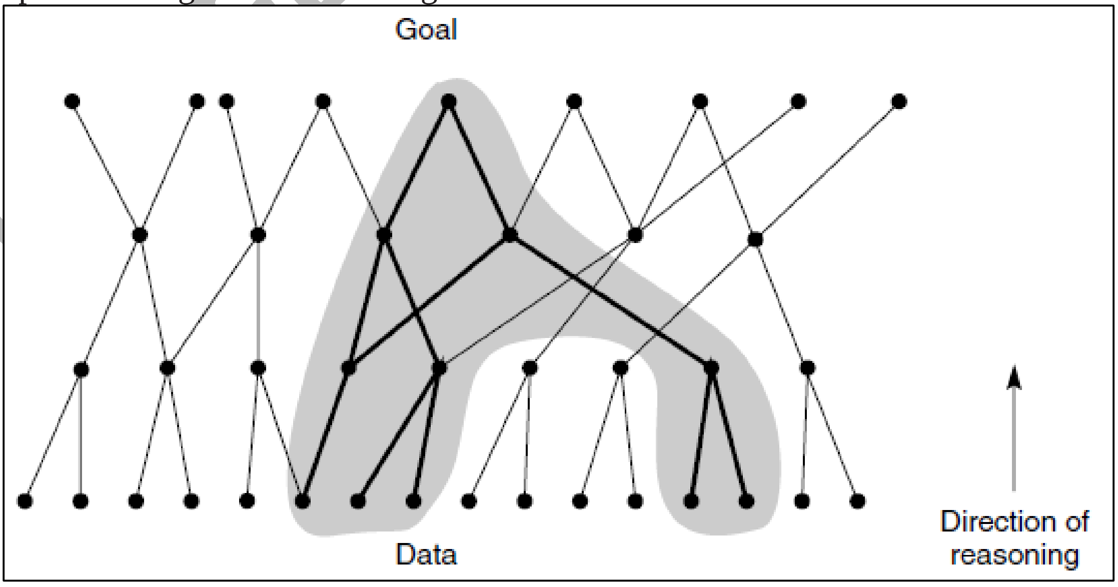


Figure (5-12) State space in which data-directed search prunes irrelevant data and their consequents and determines one of a number of possible goals.

To summarize, there is no substitute for careful analysis of the particular problem to be solved, considering such issues as the *branching factor* of rule applications how many new states are generated by rule applications in both directions?, availability of data, and ease of determining potential goals.

5.3 SEARCHING TECHNIQUES

As we have already discussed that searching our state space forms the core heart of the AI problem solving. Once a problem is formulated, we need then to solve it. We can use these techniques in many areas like theorem proving, game playing, expert system, natural language processing etc. This involves the tasks like deduction, inference, planning, common sense reasoning etc.

5.3.1 Types of search

As we know in the search methods or techniques, we firstly select one option and leave the other options if this option is our final goal, (solution) else we continue selecting, testing and expanding until either solution is found or there are no more state to be expanded. This will be determined by the search methods, so we need many different types of search algorithms. Basically there are two types of searches:

1. Uniformed search or blind search or unguided.
2. Informed or heuristic search or guided.

But please not that all search techniques are distinguished by the order in which nodes are expanded. Various search strategies are shown in figure 5-13

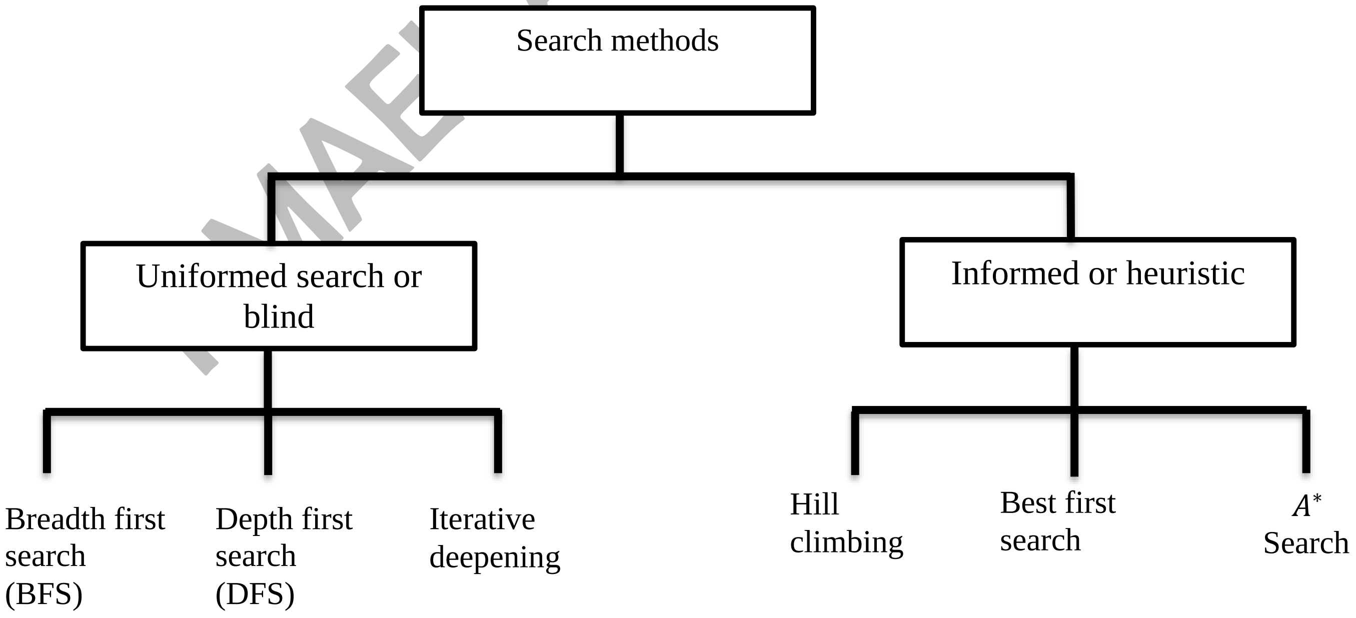


Figure (5-13) sub area of search types

All of these search strategies have advantage and drawbacks based not on the types of the problems and it representation but also on the computer resources that available to solve because each mechanism of search have its own consideration to manage time and space complexity.

5.3.1.1 Breadth first search (BFS)

It is the simplest form of the blind search. In this technique the root expand first, then all its successors are expanded and then their successors and so on. **In general, in BFS all nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.** Search tree generated by BFS shown in figure (5-14).

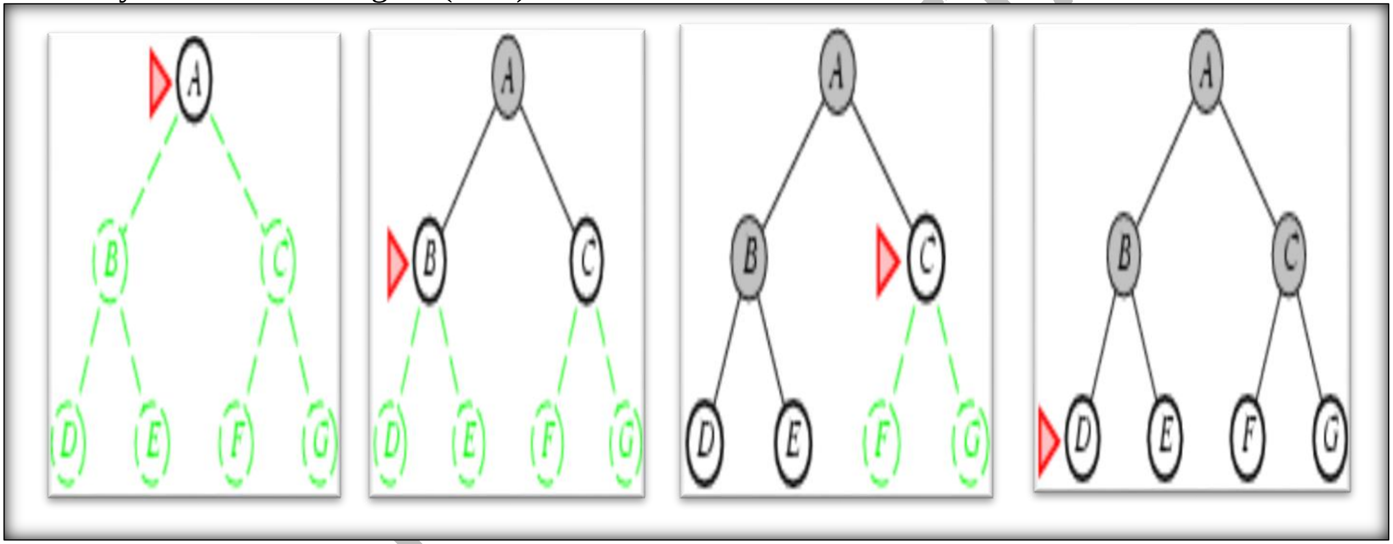


Figure (5-14) Breadth first search on a simple binary tree

We implement breadth-first search using lists, **open** and **closed**, to keep track of progress through the state space. Open, lists states that have been generated but whose children have not been examined. The order in which states are removed from open determines the order of the search. Closed records states already examined.

```
function breadth_first_search;

begin
  open := [Start];                                              % initialize
  closed := [ ];
  while open ≠ [ ] do                                           % states remain
    begin
      remove leftmost state from open, call it X;
        if X is a goal then return SUCCESS                      % goal found
          else begin
            generate children of X;
            put X on closed;
            discard children of X if already on open or closed;   % loop check
            put remaining children on right end of open             % queue
          end
    end
  return FAIL                                                   % no states left
end.
```

Child states are generated by inference rules, legal moves of a game, or other state transition operators. Each iteration produces all children of the state X and adds them to open. Note that open is maintained as a ***queue***, or first-in-first-out **(FIFO)** data structure. States are added to the right of the list and removed from the left. This biases search toward the states that have been on open the longest, causing the search to be breadth-first. Child states that have already been discovered (already appear on either open or closed) are discarded. If the algorithm terminates because the condition of the "while" loop is no longer satisfied (open = [ ]) then it has searched the entire graph without finding the desired goal: the search has failed.

Time and space complexity of (BFS)

The amount of time taken for generating these nodes is proportional to the depth, **d** and branching factor, **b** and is given by:

$$\boldsymbol{1 + b + b^2 + b^3 + \cdots + b^d + b^{d+1} - b = O}(\boldsymbol{b^{d+1}})$$

Every state has **b** successors. The root of the search tree generates **b** nodes at the first level, each of which generates more **b** nodes, for total b^2 at the second level, b^3 at the third level and so on. Suppose the solution at depth **d** in the worst case we expand all but the last nod at level **d** (since the goal itself not expanded), generating $b^{d+1} - b$ at level$d + 1$, that's how the above equation construct. Thus we have space complexity in the following order $\boldsymbol{O}\ (\boldsymbol{b^d})$.

Advantage of BFS

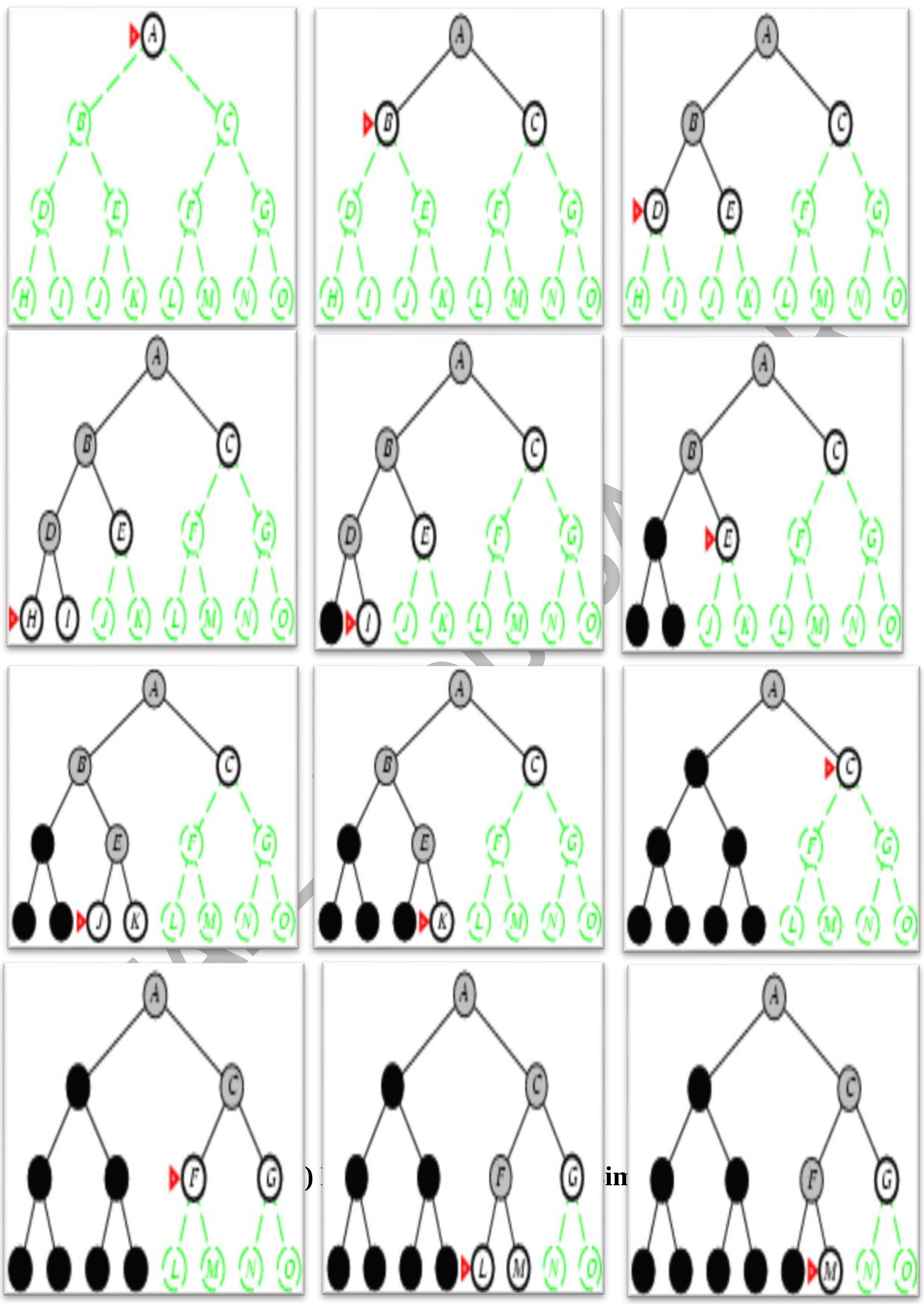
1. BFS will never get trapped exploring a blind alley
2. It is guaranteed to find a solution if one exists.

Disadvantage of BFS

1. Time complexity and space complexity are both exponential type, this is big hurdle.
2. All nodes are to be generated in BFS. So, even unwanted nodes are to be remembered (stored in queue) which is of no practical use of the search.

<u>Depth first search</u>

The descendant states are added and removed from the left end of open: open is maintained as a stack, or *last-in-first-out* (**LIFO**) structure. The organization of open as a stack directs search toward the most recently generated states, producing a depth-first search order. Search tree generated by BFS shown in figure (5-15).


A
B
C
D
E
F
G
H
I
J
K
L
M
N
O

```
function depth_first_search;

begin
  open := [Start];                                              % initialize
  closed := [ ];
  while open ≠ [ ] do                                          % states remain
    begin
      remove leftmost state from open, call it X;
      if X is a goal then return SUCCESS                          % goal found
        else begin
          generate children of X;
          put X on closed;
          discard children of X if already on open or closed;     % loop check
          put remaining children on left end of open                 % stack
        end
    end;
  return FAIL                                                  % no states left
end.
```

Time and space complexity of (DFS)
The amount of time taken for generating these nodes is proportional to the depth, **d** and branching factor, **b** and is given by: $\boldsymbol{O}\ (\boldsymbol{b^d})$.
And space complexity given by linear function of depth, d. So,
Space complexity $= O(db)$.

Advantage of DFS
1. Memory requirements in DFS are less as only nodes on the current path are stored.
2. By chance, DFS may find a solution without examining much of search space of all.

Disadvantage of DFS
This type of search can go on and on, deeper and deeper into the search space and thus, we can get lost. This referred to as *blind alley*.
<u>**EXAMPLE OF APPLING BFS & DFS**</u>
Let us consider the tree in the figure (5-16): ***first*** will apply *BFS* : U is the goal

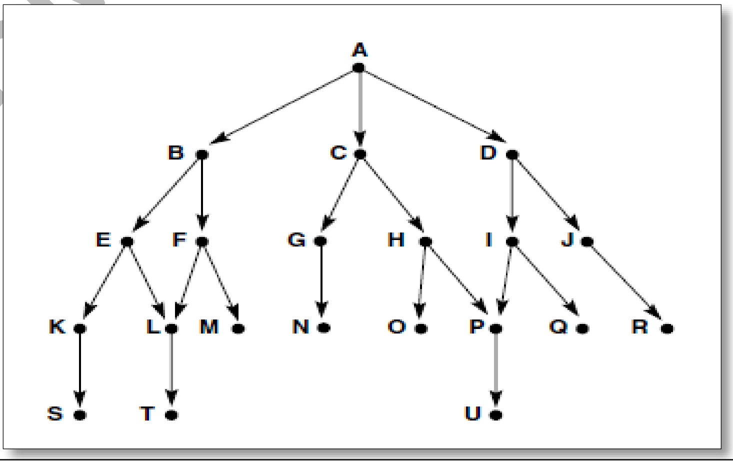


Figure (5-16) Graph for breadth- and depth-first search examples.

1. open = [A]; closed = [ ]
2. open = [B,C,D]; closed = [A]
3. open = [C,D,E,F]; closed = [B,A]
4. open = [D,E,F,G,H]; closed = [C,B,A]
5. open = [E,F,G,H,I,J]; closed = [D,C,B,A]
6. open = [F,G,H,I,J,K,L]; closed = [E,D,C,B,A]
7. open = [G,H,I,J,K,L,M] (as L is already on open); closed = [F,E,D,C,B,A]
8. open = [H,I,J,K,L,M,N]; closed = [G,F,E,D,C,B,A]
9. and so on until either U is found or open = [ ].

Figure 5.17 illustrates the graph of Figure 5.16 after six iterations of breadth_first_search. The states on open and closed are highlighted. States not shaded have not been discovered by the algorithm. Note that open records the states on the "frontier" of the search at any stage and that closed records states already visited.

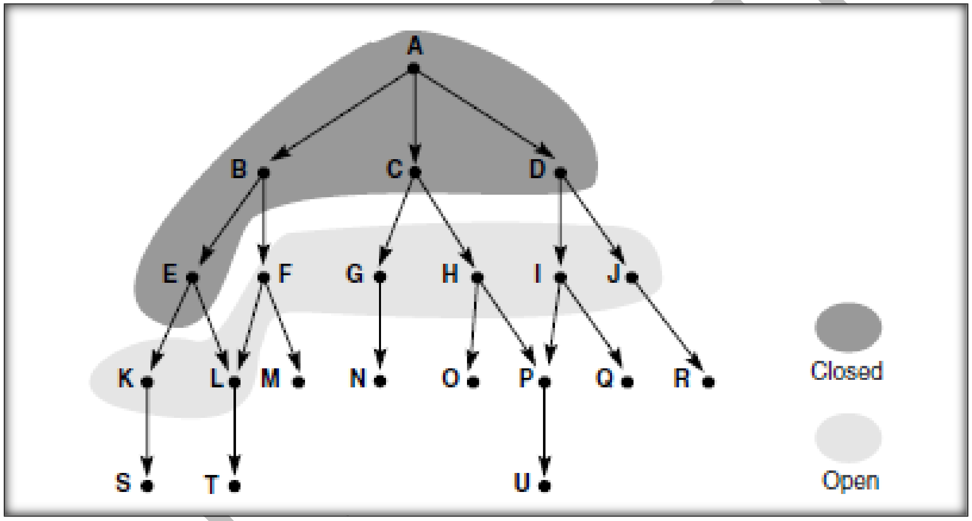


Figure (5-17) Graph of Figure (5-16) at iteration 6 of breadth-first search. States on open and closed are highlighted.

A trace of ***depth_first_search*** on the graph of Figure (5-16) appears below. The initial states of open and closed are given on line 1. Assume U is the goal state.
1. open = [A]; closed = [ ]
2. open = [B,C,D]; closed = [A]
3. open = [E,F,C,D]; closed = [B,A]
4. open = [K,L,F,C,D]; closed = [E,B,A]
5. open = [S,L,F,C,D]; closed = [K,E,B,A]
6. open = [L,F,C,D]; closed = [S,K,E,B,A]
7. open = [T,F,C,D]; closed = [L,S,K,E,B,A]
8. open = [F,C,D]; closed = [T,L,S,K,E,B,A]
9. open = [M,C,D], (as L is already on closed); closed = [F,T,L,S,K,E,B,A]
10. open = [C,D]; closed = [M,F,T,L,S,K,E,B,A]
11. open = [G,H,D]; closed = [C,M,F,T,L,S,K,E,B,A]
and so on until either U is discovered or ***open = [ ].***

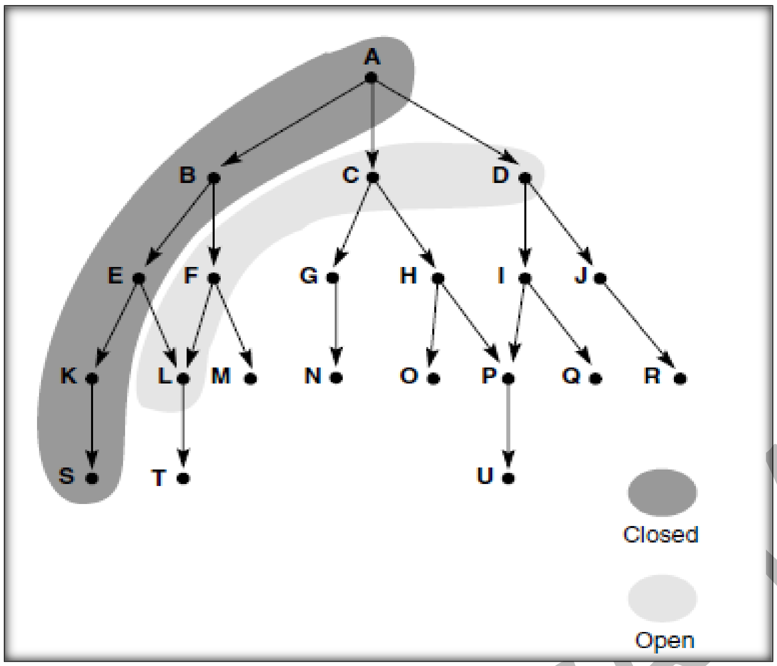


Figure (5-18) Graph of Figure (5-16) at iteration 6 of depth-first search. States on open and closed are highlighted.