Unit 3 Problem Solving and Search

Structure:

3.1 Introduction

In the last unit, we have discussed the foundations and principles of Artificial Intelligence along with major contemporary advances in Artificial Intelligence. In this unit we shall focus on problem solving and search techniques. We need to do the following for building a system that solves a particular problem: Defining the problem precisely, analysis of the problem, knowledge representation and choosing the best problem-solving techniques and apply them to the particular problem. Here in this unit we shall discuss in detail all these except knowledge representation which is better discussed in the units 5-7.

Objectives:

After studying this unit, you should be able to

- describe state space search with examples
- explain production system, rules and control strategies
- list the advantages and dis advantages of Breadth-First and Depth-First search
- explain Heuristic Search
- list problem characteristics.

3.2 State Space Search

State space search is a process used in the field of artificial intelligence (AI) in which successive configurations or *states* of an instance are considered, with the goal of finding a *goal state* with a desired property. The concept of State Space Search is widely used in Artificial Intelligence. The idea is that a problem can be solved by examining the steps which might be taken towards its solution. Each action takes the solver to a new state.

In AI, problems are often modeled as a state space, a set of *states* that a problem can be in. The set of states form a graph where two states are connected if there is an *operation* that can be performed to transform the first state into the second.

State space search as used in AI differs from traditional computer science search methods because the state space is *implicit*: the typical state space graph is much too large to generate and store in memory. Instead, nodes are generated as they are explored, and typically discarded thereafter. A solution to a combinatorial search instance may consist of the goal state itself, or of a path from some *initial state* to the goal state.

A state space consists of:

- A representation of the *states* the system can be in. In a board game, for example, the board represents the current state of the game.
- A set of *operators* that can change one state into another state. In a board game, the operators are the legal moves from any given state. Often the operators are represented as programs that change a state representation to represent the new state.
- An *initial state*.
- A set of *final states*; some of these may be desirable, others undesirable. This set is often represented implicitly by a program that detects terminal states.

The classic example is of the Farmer who needs to transport a Chicken, a Fox and some Grain across a river one at a time. The Fox will eat the Chicken if left unsupervised. Likewise the Chicken will eat the Grain.

In this case, the State is described by the positions of the Farmer, Chicken, Fox and Grain. The solver can move between States by making a legal

move (which does not result in something being eaten). Non-legal moves are not worth examining.

The solution to such a problem is a list of linked States leading from the Initial State to the Goal State. This may be found either by starting at the Initial State and working towards the Goal state or vice-versa.

The required State can be worked towards by either:

- Depth-First Search: Exploring each strand of a State Space in turn.
- Breadth-First Search: Exploring every link encountered, examining the state space a level at a time.

These techniques generally use lists of:

- Closed States: States whose links have all been explored.
- Open States: States which have been encountered, but have not been fully explored.

Ideally, these lists will also be used to prevent endless loops.

Defining the problem as a state space search

In order to solve the problem in playing a game, which is restricted to two person table or board games, we require the rules of the game and the targets for winning as well as a means of representing positions in the game. The opening position can be defined as the initial state and a winning position as a goal state, there can be more than one. Legal moves allow for transfer from initial state to other states leading to the goal state. However the rules are far too copious in most games especially chess where they exceed the number of particles in the universe. Thus the rules cannot in general be supplied accurately and computer programs cannot easily handle them. The storage also presents another problem but searching can be achieved by hashing.

The number of rules that are used must be minimized and the set can be produced by expressing each rule in as general a form as possible. The representation of games in this way leads to a state space representation and it is natural for well organized games with some structure. This representation allows for the formal definition of a problem which necessitates the movement from a set of initial positions to one of a set of target positions. It means that the solution involves using known techniques and a systematic search. This is quite a common method in AI.

3.2.1 Tic-tac-toe as a state space

State spaces are good representations for board games such as Tic-Tac-Toe. The state of a game can be described by the contents of the board and the player whose turn is next. The board can be represented as an array of 9 cells, each of which may contain an X or O or be empty.

- **State:**
  - Player to move next: X or O.
  - Board configuration:

X		O
	O	
X		X

- **Operators:** Change an empty cell to X or O.
- **Start State:** Board empty; X's turn.
- **Terminal States:** Three X's in a row; Three O's in a row; All cells full.

3.2.2 A water jug problem

You are given two jugs, a 4-gallon one and a 3-gallon one (refer figure 3.1). Neither has any measuring markers on it. There is a pump that can be used to fill the jugs with water. How can you get exactly 3 gallon of water into the 4-gallon jug?

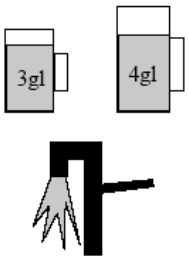


Figure 3.1: Water jug problem

State Space: The state space for this problem can be described as the set of ordered pair of integers (x,y), such that

x=0,1,3,3, or 4

and y= 0,1,3, or 3

Here x represents the number of gallons of water in the 4-gallon jug, and y represents the number of gallons of water in the 3-gallon jug.

Start State: The start state is (0,0)
Goal State: The goal state is (3,n) for any value of n (since the problem does not specify how many gallons need to be in the 3-gallon jug.

The operators to be used to solve the problem can be described as shown next. They are represented as rules whose left sides are matched against the current state and whose right sides describe the new state that results from applying the rule. In order to describe the operators completely, it was necessary to make explicit assumptions not mentioned in the problem statement.

Explicit Assumptions:
- We can fill a jug from the pump.
- We can pour water out of a jug onto the ground.
- We can pour water from one jug to another.
- There are no other measuring devices available.

Additional assumptions such as these are almost always required when converting from a typical problem statement given in English to a formal representation of the problem suitable for use by a program.

The major production rules for solving this problem are shown below:

Initial condition	Goal	Comment
1. (x,y), if x < 4	(4,y)	Fill 4-gallon jug from tap
3. (x,y), if y < 3	(x,3)	Fill 3-gallon jug from tap
3. (x,y), if x > 0	(x-d,y)	Pour some water out of the 4-gallon jug
4. (x,y), if y > 0	(x,y-d)	Pour some water out of the 3-gallon jug
5. (x,y), if x > 0	(0,y)	Empty the 4-gallon jug on the ground
6. (x,y), if y > 0	(x,0)	Empty the 3-gallon jug on the ground
7. (x,y), if x+y ≥ 4 & y > 0	(4,y-(4-x))	Pour water from the 3-gallon jug into the 4-gallon jug until the 4-gallon jug is full
8. (x,y), if x+y ≥ 3 & x > 0	(x-(3-y),3)	Pour water from the 4-gallon jug into the 3-gallon jug until the 3-gallon jug is full
9. (x,y), if x+y ≤ 4 & y > 0	(x+y,0)	Pour all the water From the 3-gallon jug into the 4-gallon jug
10. (x,y), if x+y ≤ 3 & x > 0	(0,x+y)	Pour all the water From the 4-gallon jug into the 3-gallon jug
11. (0,3)	(3,0)	Pour the 3 gallons from the 3-gallon jug into the 4-gallon jug.
13. (3,y)	(0,y)	Empty the 3 gallons in the 4-gallon jug on the ground and a solution is given below:

Gallons in the 4-gallon jug	Gallons in the 3-gallon jug	Rule applied
0	0	3
0	3	9
3	0	3
3	3	7
4	3	5 or 13
0	3	9 or 11
3	0	

The control strategy for selecting the rules has not yet been fully discussed but if all the rules are applied to each of the states those that are applicable will produce new states and the required goal state can be searched for.

Self Assessment Questions

1. State space search is a process used in the field of _____ in which successive configurations or *states* of an instance are considered, with the goal of finding a *goal state* with a desired property.
2. State space search used in AI is same as traditional computer science search methods. (State True or False)
3. _____ are good representations for board games such as Tic-Tac-Toe.

3.3 Production System

A **production system** (or **production rule system**) is a computer program typically used to provide some form of artificial intelligence, which consists primarily of a set of rules about behavior. These rules, termed **productions**, are basic representations which are very useful in AI planning, expert systems and action selection. A production system provides the mechanism necessary to execute productions in order to achieve some goal for the system.

A Production System consists of:

1) A set of rules, each consisting of two parts: a sensory precondition (or “IF’’ statement) and an action (or “THEN’’). If a production's precondition matches the current state of the world, then the production is said to be *triggered*. If a production's action is executed, it is said to have *fired*.
2) One or more knowledge/databases, sometimes called working memory, which maintains data about current state or knowledge. Some part of the

database may be permanent, while other parts of it may pertain only to the solution of the current problem. The information in these databases may be structured in any appropriate way.

3) A control strategy that specifies the order in which the rules will be compared to the database and a way of resolving the conflicts that arise when several rules match at once.
4) A rule interpreter/applier. The rule interpreter must provide a mechanism for prioritizing productions when more than one is triggered.

Basic operation

Rule interpreters generally execute a forward chaining, given next, algorithm for selecting productions to execute to meet current goals, which can include updating the system's data or beliefs. The condition portion of each rule (*left-hand side* or LHS) is tested against the current state of the working memory.

In idealized or data-oriented production systems, there is an assumption that any triggered conditions should be executed: the consequent actions (*right-hand side* or RHS) will update the agent's knowledge, removing or adding data to the working memory. The system stops processing either when the user interrupts the forward chaining loop; when a given number of cycles have been performed; when a “halt’’ RHS is executed, or when no rules have true LHSs.

Real-time and expert systems, in contrast, often have to choose between mutually exclusive productions --- since actions take time, only one action can be taken, or (in the case of an expert system) recommended. In such systems, the rule interpreter, or inference engine, cycles through two steps: matching production rules against the database, followed by choosing which of the matched rules to apply and executing the selected actions.

Matching production rules against working memory

Production systems may vary on the expressive power of conditions in production rules. Accordingly, the pattern matching algorithm which collects production rules with matched conditions may range from the naive -- trying all rules in sequence, stopping at the first match -- to the optimized, in which rules are “compiled’’ into a network of inter-related conditions.

The latter is illustrated by the RETE algorithm, designed by Charles L. Forgy in 1983, which is used in a series of production systems, called OPS and originally developed at Carnegie Mellon University culminating in OPS5 in the early eighties. OPS5 may be viewed as a full-fledged programming language for production system programming.

Choosing which rules to evaluate/Conflict resolution

Production systems may also differ in the final selection of production rules to execute, or *fire*. The collection of rules resulting from the previous matching algorithm is called the *conflict set*, and the selection process is also called a *conflict resolution strategy*.

Here again, such strategies may vary from the simple – use the order in which production rules were written; assign weights or priorities to production rules and sort the conflict set accordingly – to the complex – sort the conflict set according to the times at which production rules were previously fired; or according to the extent of the modifications induced by their RHSs. Whichever conflict resolution strategy is implemented, the method is indeed crucial to the efficiency and correctness of the production system.

Most conflict resolution schemes are very simple, dependent on the number of conditions in the production, the time stamps (ages) of the elements to which the conditions matched, or completely random. One of the advantages of production systems is that the computational complexity of the matcher, while large, is deterministically finite and the conflict resolution scheme is trivial. This is in contrast to logicist systems in which declarative knowledge may be accessed instantly but the time required to use the knowledge (in a theorem prover, for instance) can not be pre-determined.

Actions

The actions of productions are manipulations to working memory. Elements may be added, deleted and modified. Since elements may be added and deleted, the production system is *non-monotonic*: the addition of new knowledge may obviate previous knowledge. Non-monotonicity increases the significance of the conflict resolution scheme since productions which match in one cycle may not match in the following because of the action of the intervening production. Some production systems are *monotonic*, however, and only add elements to working memory, never deleting or

modifying knowledge through the action of production rules. Such systems may be regarded as implicitly parallel since all rules that match will be fired regardless of which is fired first.

Using production systems

The use of production systems varies from simple string rewriting rules to the modeling of human cognitive processes, from term rewriting and reduction systems to expert systems.

3.3.1 Rules

These are also called **condition-action rules**. These components of a rule-based system have the form:

if <condition> then <conclusion>

or

if <condition> then <action>

Example:

if a patient has high levels of the enzyme ferritin in his blood **and** a patient has the Cys383→Tyr mutation in HFE gene **then** conclude patient has haemochromatosis

Rules can be evaluated by:

- backward chaining
- forward chaining

Backward Chaining

- To determine if a decision should be made, work backwards looking for justifications for the decision.
- Eventually, a decision must be justified by facts.

The figure 3.2 shows the concept of Backward Chaining

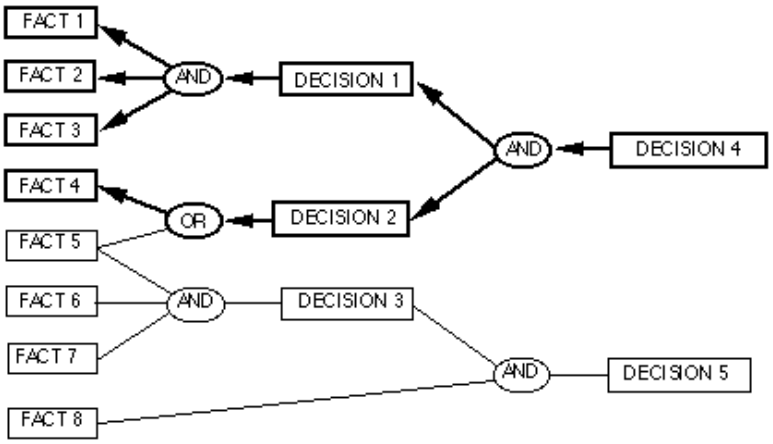


Figure 3.2: Backward Chaining

Forward Chaining

Now let us see how Rules can be evaluated by Forward Chaining

- Given some facts, work forward through inference net.
- Discovers what conclusions can be derived from data.

The figure 3.3 shows the concept of Forward Chaining

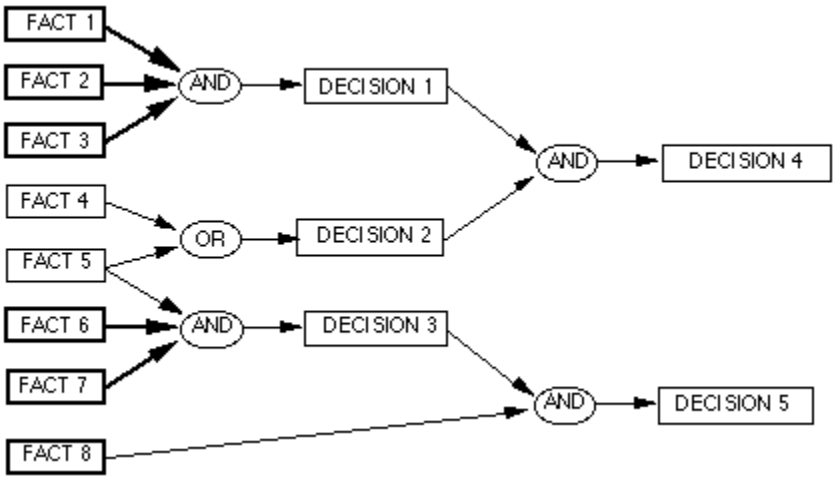


Figure 3.3: Forward Chaining

Until a problem is solved or no rules 'if' part is satisfied by the current situation:

1) Collect rules whose 'if' parts are satisfied.
2) If more than one rules 'if' part is satisfied, use a conflict resolution strategy to eliminate all but one.

3) Do what the rules 'then' part says to do.

A simple example of production system (String rewriting)

This example shows a set of production rules for reversing a string from an alphabet that does not contain the symbols "$" and "*" (which are used as marker symbols).

P1: $$ -> *
P3: *$ -> *
P3: *x -> x*
P4: * -> null & halt
P5: $xy -> y$x
P6: null -> $

In this example, production rules are chosen for testing according to their order in this production list. For each rule, the input string is examined from left to right with a moving window to find a match with the LHS of the production rule. When a match is found, the matched substring in the input string is replaced with the RHS of the production rule. In this production system, x and y are *variables* matching any character of the input string alphabet. Matching resumes with P1 once the replacement has been made.

The string “ABC’’, for instance, undergoes the following sequence of transformations under these production rules:

$ABC (P6)
B$AC (P5)
BC$A (P5)
BCA (P6)
CBA (P5)
CB$A (P6)
$$CBA (P6)
*CBA (P1)
C*BA (P3)
C*B$A (P3)
CB*$A (P3)
CB*A (P3)
CBA* (P3)
CBA (P4)

In such a simple system, the ordering of the production rules is crucial. Often, the lack of control structure makes production systems difficult to design. It is, of course, possible to add control structure to the production systems model, namely in the inference engine, or in the working memory. Figure 3.4: shows the cost of Production System.

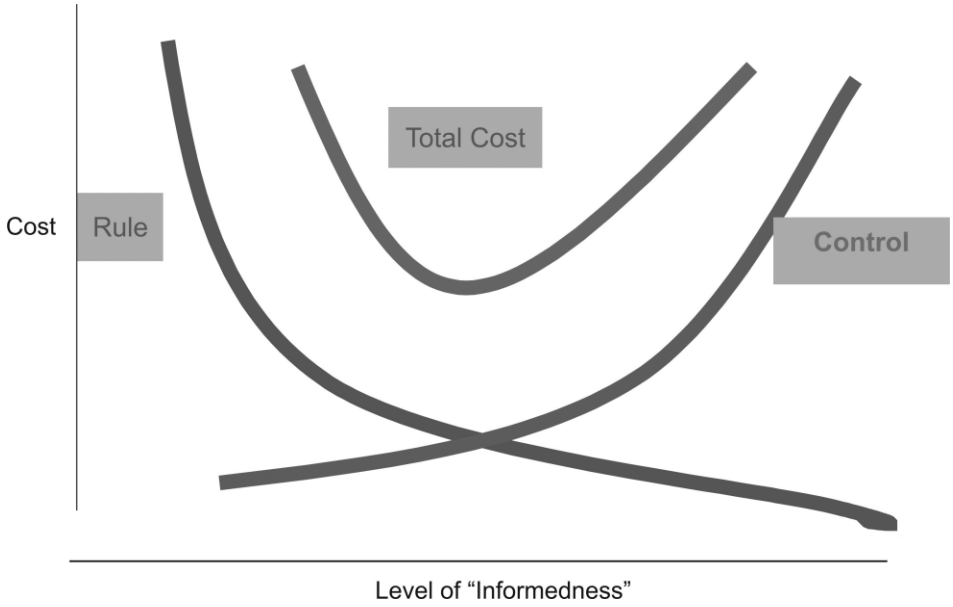


Figure 3.4: Cost of Production System

3.3.2 Control strategies

So far we have not discussed, fully, the question of how to decide which rule to apply next during the process of searching for a solution of the problem. This question arises since, often, more than one rule (and sometimes fewer than one rule) will have its left side match the current state. Even without a great deal of thought, it is clear that how such decisions are made will have a crucial impact on how quickly, and even whether, a problem is finally solved.

1) The first requirement of a good control strategy is that it cause motion. Consider again the water jug problem. Suppose we implemented the simple control strategy of starting each time at the top of the list of rules and choosing the first applicable one. If we did that, we would never solve the problem. We would continue indefinitely filling the 4-gallon jug with water. Control strategies that do not cause motion will never lead to a solution.

2) The second requirement of a good control strategy is that it be systematic. Here is another simple control strategy for the water jug problem: On each cycle, choose at random from among the applicable rules. This strategy is better than the first. It causes motion. It will lead to a solution eventually. But we are likely to arrive at the same state several times during the process and to use many more steps than are necessary. Because the control strategy is not systematic, we may explore a particular useless sequence of operators several times before we finally find a solution. The requirement that a control strategy be systematic corresponds to the need for global motion (over the course of several steps) as well as for local motion (over the course of a single step). One systematic control strategy for the water jug problem is the following. Construct a tree with the initial state as its root. Generate all the offspring of the root by applying each of the applicable rules to the initial state. Now for each leaf node, generate all its successors by applying all the rules that are appropriate. Continue this process until some rule produces a goal state. This process, called breadth-first search, can be described as follows:

Breadth-first search

Breadth-first searches are performed by exploring all nodes at a given depth before proceeding to the next level. This means that all immediate children of the nodes are explored before any of the children's children are considered. It has the obvious advantage of always finding a minimal path length solution when one exists. However, a great many nodes may need to be explored before a solution is found, especially if the tree is very full.

Time complexity of breadth-first search: The time complexity of Breadth-First Search is $O(b^d)$. This can be seen by noting that all nodes upto the goal depth d are generated. Therefore, the number generated is $1 + b + b^3 + \ldots + b^d$, which is $O(b^d)$.

Space complexity of breadth-first search: The space complexity of Breadth-First Search is also $O(b^d)$ since all nodes at a given depth must be stored in order to generate the nodes at the next depth, that is, b^{d-1} nodes must be stored at depth d-1 to generate nodes at depth d, which gives space complexity of $O(b^d)$. This can be seen by noting that all nodes upto

the goal depth d are generated. Therefore, the number generated is $1 + b + b^3 + \ldots + b^d$, which is $O(b^d)$.

This is not good. For example consider the following table for b=10, 1 node/ms, 100 bytes/node:

Table 3.1: Depth versus time and memory requirements

Depth	Nodes	Time	Memory
0	1	1 millisecond	100 bytes
2	111	.1 seconds	11 kilobytes
4	11,111	11 seconds	1 megabyte
6	10^6	18 minutes	111 megabytes
8	10^8	31 hours	11 gigabytes
10	10^{10}	128 days	1 terabyte
12	10^{12}	35 years	111 terabytes
14	10^{14}	3500 years	11,111 terabytes

Note that memory requirements are a bigger problem here than execution time.

In general, exponential complexity search problems cannot be solved for any but the smallest instances.

Advantages: The Advantages of breadth-first search are:

- Always finds the goal
- Good for shallow trees
- If there is a solution, then breadth-first search is guaranteed to find it. Furthermore, if there are multiple solutions, then a minimal solution (i.e. one that requires the minimum number of steps) will be found. This is guaranteed by the fact that longer paths are never explored until all shorter ones have already been examined.

Disadvantages: The disadvantages of breadth-first search are:

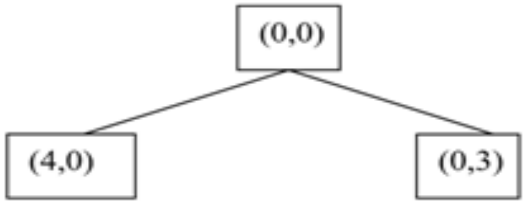
- Often can be slow in deep trees
- Can result in an exhaustive search
- Use of both exponential time and space is one of the main drawback

Algorithm: Breadth-first search:

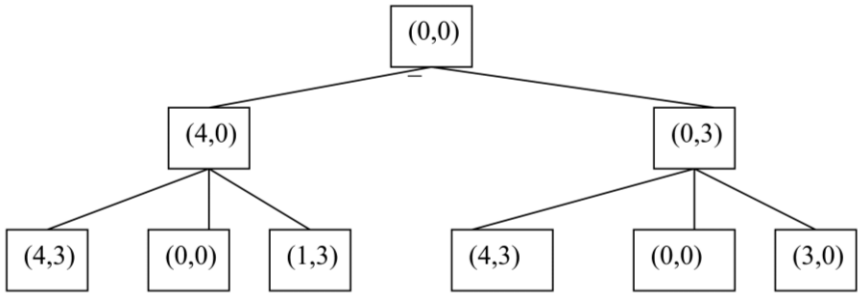
1. Set *L* to be a list of the initial nodes in the problem.
2. If *L* is empty, *fail* otherwise pick the first node *n* from *L*
3. If *n* is a goal state, quit and return path from initial node.

4. Otherwise remove *n* from *L* and add to the end of *L* all of *n*'s children. Label each child with its path from initial node. Return to 3.

The figure 3.5 shows the one and two levels of Breadth-first search tree for water jug problem.



(a) One level of Breadth-first search tree



(b) Two level of Breadth-first search tree

Figure 3.5: Breadth-first search tree for Water Jug problem

Other systematic control strategies are also available. For example, we could pursue a single branch of the tree until it yields a solution or until a decision to terminate the path is made. It makes sense to terminate a path if it reaches a dead-end, produces a previous state, or becomes longer than some pre-specified “futility” limit. In such a case, backtracking occurs. The most recently created state, from which alternatives moves are available, will be revisited and a new state will be created. This form of backtracking is called chronological backtracking because the order in which steps are undone depends only on the temporal sequence in which the steps were originally made. Specifically, the most recent step is always the first to be undone. This search procedure is also called depth-first search and is explained as follows:

Depth-first search

Depth-first searches are performed by diving downward into a tree as quickly as possible. It does this by always generating a child node from the most recently expended node, then generating that child's children, and so on until a goal is found or some cutoff point d is reached. If a goal is not found when a leaf node is reached or at the cutoff point, the program backtracks to the most recently expended node and generates another of its children. This process continues until a goal is found or failure occurs. This search is very reliable but sometimes can take a long time or even turn into an exhaustive search (a search that looks at every point until it reaches the goal). The Depth-First search can also be modified to go from the right to the left.

Time complexity of depth-first search: The time complexity of Depth-First search is $O(b^d)$.

Space complexity of depth-first search: The space complexity of Depth-First search is O(d) since only the path from the starting node to the current node need to be stored. Therefore, if the depth cutoff is d, the space complexity is just O(d).

- ***Advantages:*** The Advantages of Depth-first search are:
  Always finds the goal
- The depth-first search is preferred over the breadth-first search when the search tree is known to have a plentiful number of goals.
- It requires less memory as space complexity is O(d) compared to $O(b^d)$ of breadth-first search.

Disadvantages: The Disadvantages of Depth-first search are:

- Often can be slow
- Can result in an exhaustive search
- Can get stuck in long paths that don't reach the goal
- The depth cutoff introduces some problems. If it is set too shallow, goals may be missed; if it is set too deep, extra computation may be performed.

Algorithm: Blind search depth-first search:

1. Set *L* to be a list of the initial nodes in the problem.
2. If *L* is empty, *fail* otherwise pick the first node *n* from *L*
3. If *n* is a goal state, quit and return path from initial node.

4. Otherwise remove *n* from *L* and add to the front of *L* all of *n*'s children. Label each child with its path from initial node. Return to 3.

Figure 3.6 shows Depth-first and Breadth-first search.

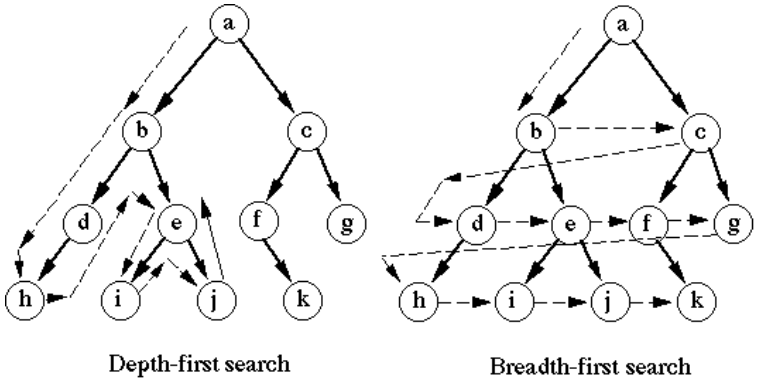
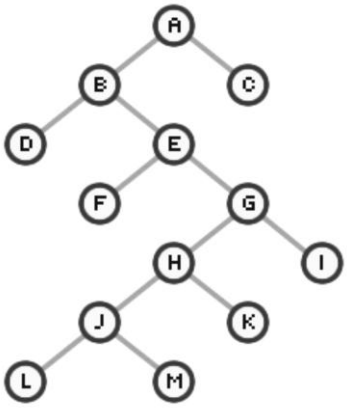


Figure 3.6: Depth-first and Breadth-first search

Example: Depth First Search

Let's find a path between nodes A and F:



Step 0:

Let's start with our root/goal node:

We shall be using two lists to keep track of what we are doing - an **Open** list and a **Closed List**. An Open list keeps track of what you need to do, and the Closed List keeps track of what you have already done. Right now, we

only have our starting point, node A. We haven't done anything to it yet, so let's add it to our **Open** list.

- Open List: A
- Closed List: <empty>

Step 1:

Now, let's explore the neighbors of our A node. To put it another way, let's take the first item from our Open list and explore its neighbors:

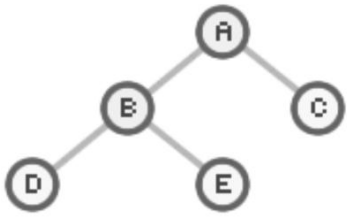
Node A's neighbors are the B and C nodes. Because we are now done with our A node, we can remove it from our Open list and add it to our Closed List. We have not done with this step though. We now have two new nodes B and C that need exploring. Add those two nodes to our Open list.

Our current Open and Closed Lists contain the following data:

- Open List: B, C
- Closed List: A

Step 2:

Our Open list contains two items. For depth first search and breadth first search, we always explore the first item from our Open list. The first item in our Open list is the B node. B is not our destination, so let's explore its neighbors:



Because we have now expanded B, we are going to remove it from the Open list and add it to the Closed List. Our new nodes are D and E, and we add these nodes to the ***beginning*** of our Open list:

- Open List: D, E, C
- Closed List: A, B

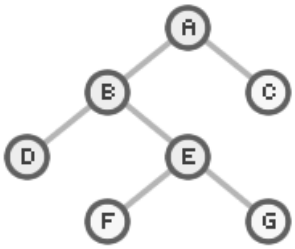
Step 3:

Because D is at the beginning of our Open List, we expand it. D isn't our destination, and it does not contain any neighbors. All we do in this step is to remove D from our Open List and add it to our Closed List:

- Open List: E, C
- Closed List: A, B, D

Step 4:

We now expand the E node from our Open list. E is not our destination, so we explore its neighbors and find out that it contains the neighbors F and G. Remember, F is our target, but we don't stop here though. Despite F being on our path, we only end when we are about to ***expand*** our target Node - F in this case:

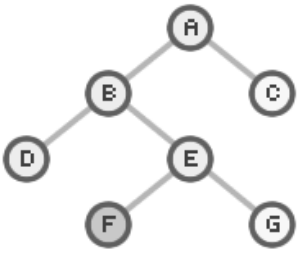


Our Open list will have the E node removed and the F and G nodes added. The removed E node will be added to our Closed List:

- Open List: F, G, C
- Closed List: A, B, D, E

Step 5:

We now expand the F node. Since it is our intended destination, we stop:



We remove F from our Open list and add it to our Closed List. Since we are at our destination, there is no need to expand F in order to find its neighbors. Our final Open and Closed Lists contain the following data:

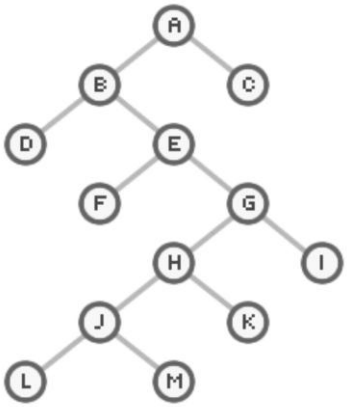
- Open List: G, C
- Closed List: A, B, D, E, F

The final path taken by our depth first search method is what the final value of our Closed List is: A, B, D, E, F.

Example: Breadth first search:

In depth first search, newly explored nodes were added to the **beginning** of our Open list. In breadth first search, newly explored nodes are added to the **end** of your Open list.

Let's see how that change will affect our results. For reference, here is our original search tree:



Let's try to find a path between nodes A and E.

Step 0:

Let's start with our root/goal node:

Like before, we will continue to employ the Open and Closed Lists to keep track of what needs to be done:

- Open List: A
- Closed List: <empty>

Step 1:

Now, let's explore the neighbors of our A node. So far, we are following in depth first's foot steps:

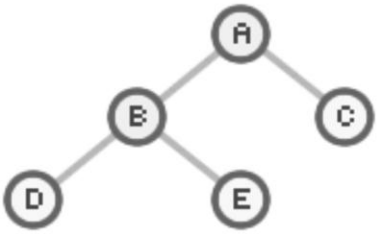
We remove A from our Open list and add A to our Closed List. A's neighbors, the B and C nodes, are added to our Open list. They are added to the end of our Open list, but since our Open list was empty (after removing A), it's hard to show that in this step.

Our current Open and Closed Lists contain the following data:

- Open List: B, C
- Closed List: A

Step 2:

Here is where things start to diverge from our depth first search method. We take a look at the B node because it appears first in our Open List. Because B isn't our intended destination, we explore its neighbors:

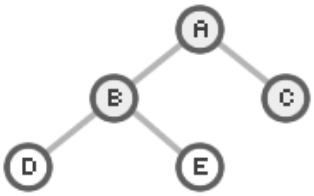


B is now moved to our Closed List, but the neighbors of B, nodes D and E are added to the ***end*** of our Open list:

- Open List: C, D, E
- Closed List: A, B

Step 3:

We now expand our C node:



Since C has no neighbors, all we do is remove C from our Closed List and move on:

- Open List: D, E
- Closed List: A, B, C

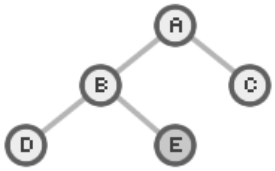
Step 4:

Similar to Step 3, we expand node D. Since it isn't our destination, and it too does not have any neighbors, we simply remove D from our to Open list, add D to our Closed List, and continue on:

- Open List: E
- Closed List: A, B, C, D

Step 5:

Because our Open list only has one item, we have no choice but to take a look at node E. Since node E is our destination, we can stop here:



Our final versions of the Open and Closed Lists contain the following data:

- Open List: <empty>
- Closed List: A, B, C, D, E

Traveling from A to E takes us through B, C, and D using breadth first search.

Traveling salesman problem (TSP)

“The traveling salesman problem, or TSP for short, is this: given a finite number of 'cities' along with the cost of travel between each pair of them, find the cheapest way of visiting all the cities and returning to your starting point.’’

It involves n cities with paths connecting the cities. A tour is any path which begins with some starting city, visits each of the other cities exactly once, and returns to the starting city. There are direct roads between each pair of cities on the list.

Objective of TSP: Objective of a traveling salesman problem is to find a minimal distance tour.

A simple, motion-causing and systematic control structure could, in principle, solve this problem. It would simply explore all possible paths in the tree and return to the one with the shortest length. This approach will even work in practice for very short lists of cities. But it breaks down quickly as the number of cities grows. If there are N cities, then the number of different paths among them is 1.3 … (N-1), or (N-1)! The time to examine a single path is proportional to N. So the total time required to perform this search is proportional to N! Assuming there are only 10 cities, 10! is 3,638,800, which is a very large number. The salesman could easily have 35 cities to visit. To solve this problem would take more time than he would be willing to spend. This phenomenon is called *combinatorial explosion.*

To combat it, we need a new control strategy. We can beat the simple strategy outlined above using a technique called *branch-and-bound.* Begin generating complete paths, keeping track of the shortest path found so far. Give up exploring any path as soon as its partial length becomes greater than the shortest path found so far. Using this technique, we are still guaranteed to find the shortest path. Unfortunately, although this algorithm is more efficient than the first one, it still requires exponential time. The exact amount of time it saves for a particular problem depends on the order in which the paths are explored. But it is still inadequate for solving large problems.

3.3.3 Heuristic search

Sometimes it is not feasible to search the whole search space - it's just too big. Imagine searching every possible road and alley in a 400 mile circumference around Delhi when looking for a route to Mumbai. (OK, so this might be feasible on a fast computer, but some search spaces are REALLY big). In this case we need to use *heuristic* search.

The basic idea of heuristic search is that, rather than trying all possible search paths, you try and focus on paths that seem to be getting you nearer your goal state. Of course, you generally can't be sure that you are really near your goal state – it could be that you'll have to take some amazingly complicated and circuitous sequence of steps to get there. But we might be able to have a good guess. Heuristics are used to help us make that guess.

Heuristics is a rule of thumb or judgmental technique that leads to a solution some of the time but provides no guarantee of success. It may in fact end in

failure. Heuristics plays an important role in search strategies because of the exponential nature of most problems. They help to reduce the number of alternatives from an exponential number to a polynomial number and, thereby, obtain a solution to a tolerable amount of time. When exhaustive search is impractical, it is necessary to compromise for a constrained search which eliminates many paths but offers the promise of success some of the time. Here, success may be considered to be finding an optimal solution a fair proportion of the time or just finding good solutions much of the time. In this regard, any policy which uses as little search effort as possible to find any qualified goal has been called a *satisfying policy*.

Thus *heuristics* is a method that

- might not always find the best solution
- ***but*** is guaranteed to find a good solution in reasonable time.
- By sacrificing completeness it increases efficiency.
- Useful in solving tough problems which
  - could not be solved any other way.
  - solutions take an infinite time or very long time to compute.

There are two cases in AI searches when heuristics is needed:

- The problem has no exact solution. For example, in medical diagnosis doctors use heuristics to choose the most likely diagnoses given a set of symptoms.
- The problem has an exact solution but is too complex to allow for a *brute force* solution.

To use heuristic search you need an *evaluation function* that scores a node in the search tree according to how close to the target/goal state it seems to be. This will just be a guess, but it should still be useful. For example, for finding a route between two towns a possible evaluation function might be a “as the crow flies’’ distance between the town being considered and the target town. It may turn out that this does not accurately reflect the actual (by road) distance - maybe there aren't any good roads from this town to your target town. However, it provides a quick way of guessing that helps in the search.

Key point: Heuristics are **fallible**. Because they rely on limited information, they may lead to a suboptimal solution or to a dead end.

Implementation: Heuristic search is implemented in two parts:

- The heuristic measure.
- The search algorithm.

Example 1: One example of a good general-purpose heuristic that is useful for a variety of combinatorial problems is called the *nearest neighbor heuristic*, which works by selecting the locally superior alternative at each step. The traveling salesman problem is too complex to be solved via exhaustive search for large values of N. The ***nearest neighbor heuristic*** works well most of the time, but with some arrangements of cities it does not find the shortest path. Applying it to the traveling salesman problem, we produce the following procedure:

1) Arbitrarily select a starting city.
2) To select the next city, look at all cities not yet visited, and select the one closest to the current city, Go to it next.
3) Repeat step 3 until all cities have been visited.

Consider the two graphs in figure 3.7. In the first, nearest neighbor will find the shortest path. In the second it does not:

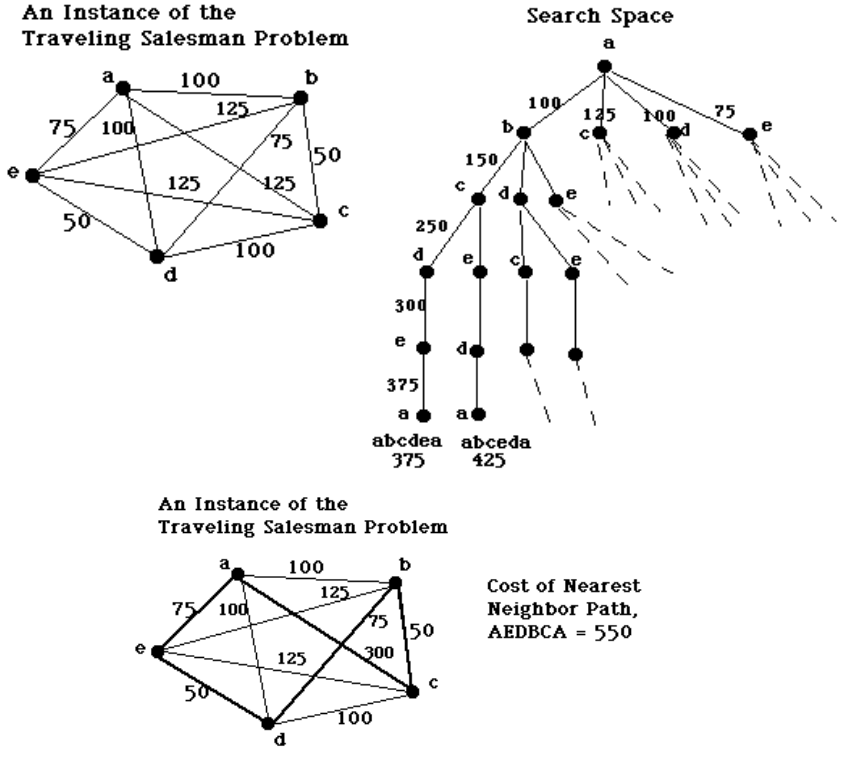


Figure 3.7: Traveling salesman problem

Thus simple heuristic for TSP is: **Move to the city which has the shortest distance among other alternatives.** It can be seen from the above example that this policy, *nearest neighbor heuristic,* gives no guarantee of an optimal solution, but its solutions are often good, and the time required is only $O(n^3)$.

Example 2: Consider the game of tic-tac-toe (refer the figure 3.8). Even if we use symmetry to reduce the search space of redundant moves, the number of possible paths through the search space is something like **13 x 7! = 60480**. That is a measure of the amount of work that would have to be done by a brute-force search.

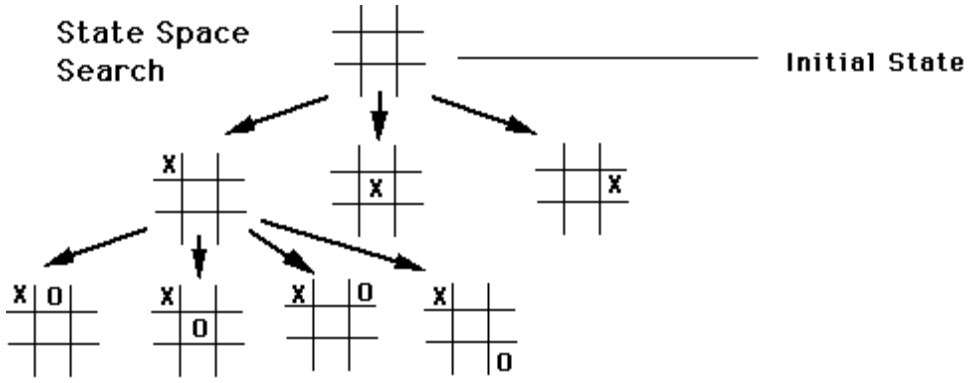


Figure 3.8: Game of tic-tac-toe

Simple heuristic for tic-tac-toe: **Move to the square in which X has the most winning lines.** Using this rule, we can see that a corner square has heuristic value of 3, a side square has a heuristic value of 3, but the center square has a heuristic value of 4. So we can ***prune*** the left and right branches of the search tree. This removes 3/3 of the search space on the first move. If we apply the heuristic at each level of the search, we will remove most of the states from consideration thereby greatly improving the efficiency of the search.

3.4 Problem Characteristics

In order to choose the most appropriate method (or combination of methods) for a particular problem, it is necessary to analyze the problem along several key dimensions:

1. Is the problem decomposable into a set of nearly independent smaller or easier sub-problems?

2. Can the solution steps be ignored or at least undone if they prove unwise?
3. Is the problem's universe predictable?
4. Is a good solution to the problem obvious without comparison to all other possible solutions?
5. Is the desired solution a state of the world or a path to a state?
6. Is a large amount of knowledge absolutely required to solve this problem or is knowledge important only to constrain the search?
7. Can a computer that is simply given the problem return the solution or will the solution of the problem require interaction between the computer and a person?

Self Assessment Questions

4. A _____ is a computer program typically used to provide some form of artificial intelligence, which consists primarily of a set of rules about behavior.
5. _____ searches are performed by diving downward into a tree as quickly as possible.
6. Heuristics is a rule of thumb or judgmental technique that leads to a solution some of the time but provides no guarantee of success. (State True or False)
7. What is the time complexity of Depth-First search?

_____

3.5 Summary

In this unit we discussed different types of problems that can be solved using artificial intelligence and different types of search techniques that can be employed for solving these problems. We also discussed production System and Heuristic search. State space search is a process used in the field of artificial intelligence (AI) in which successive configurations or states of an instance are considered, with the goal of finding a goal state with a desired property. A production system (or production rule system) is a computer program typically used to provide some form of artificial intelligence, which consists primarily of a set of rules about behavior. The actions of productions are manipulations to working memory. The first requirement of a good control strategy is that it causes motion and the second requirement is that it be systematic. Breadth-first searches are

performed by exploring all nodes at a given depth before proceeding to the next level. Depth-first search always generates a child node from the most recently expended node, then generating that child's children, and so on until a goal is found or some cutoff point d is reached. Heuristic is a rule of thumb or judgmental technique that leads to a solution some of the time but provides no guarantee of success.

3.6 Terminal Questions

1. Explain State Space Search.
2. Analyze water Jug problem.
3 What do you mean by production system? Explain briefly.
4 What are the advantages and disadvantages of Breadth-First search?
5. Write a note on Heuristic search.

3.7 Answers

Self Assessment Questions

1. Artificial intelligence (AI)
2. False
3. State spaces
4. Production system (or production rule system)
5. Depth-first
6. True
7. $O(b^d)$.

Terminal Questions

1. State space search is a process used in the field of artificial intelligence (AI) in which successive configurations or *states* of an instance are considered, with the goal of finding a *goal state* with a desired property. (Refer section 3.2 for details)
2 You are given two jugs, a 4-gallon one and a 3-gallon one. Neither has any measuring markers on it. (Refer sub-section 3.2.2)
3 A production system (or production rule system) is a computer program typically used to provide some form of artificial intelligence, which consists primarily of a set of rules about behavior. (Refer section 3.3).
4. **Advantage:** Always finds the goal
   Disadvantage: Often can be slow in deep trees
   (Refer sub-section 3.3.2)

5. The basic idea of heuristic search is that, rather than trying all possible search paths, you try and focus on paths that seem to be getting you nearer your goal state. (Refer sub-section 3.3.3)