Artificial Intelligence

Module 1

Definition

It is the science and engineering of making intelligent machines, especially intelligent computer programs. It is related to the similar task of using computers to understand human intelligence, but AI does not have to confine itself to methods that are biologically observable.

It is the study of how to make computers do things which at the moment people do better. It fails to include some areas of potentially large impact namely problems that cannot now be solved well by either computers or people.

It is the automation of activities that we associate with human thinking, activities such as decision making, problem solving, learning.

Intelligence +System → AI

Applications of AI

AI currently encompasses a huge variety of subfields, ranging from general purpose areas such as learning and perception to such specific tasks as playing chess, proving mathematical theorems, writing poetry and diagnosing diseases.

Main application areas are:

1. Game playing

You can buy machines that can play master level chess for a few hundred dollars. There is some AI in them, but they play well against people mainly through brute force computation--looking at hundreds of thousands of positions. To beat a world champion by brute force and known reliable heuristics requires being able to look at 200 million positions per second.

2. Speech recognition

In the 1990s, computer speech recognition reached a practical level for limited purposes. Thus United Airlines has replaced its keyboard tree for flight information by a system using speech recognition of flight numbers and city names. It is quite convenient. On the other hand, while it is possible to instruct some computers using speech, most users have gone back to the keyboard and the mouse as still more convenient.

3. Understanding natural language

Just getting a sequence of words into a computer is not enough. Parsing sentences is not enough either. The computer has to be provided with an understanding of the domain the text is about, and this is presently possible only for very limited domains.

4. Computer vision

The world is composed of three-dimensional objects, but the inputs to the human eye and computers' TV cameras are two dimensional. Some useful programs can work solely in two dimensions, but full computer vision requires partial three-dimensional information that is not just a set of two-dimensional views. At present there are only limited ways of representing three-dimensional information directly, and they are not as good as what humans evidently use.

5. Expert systems

A ``knowledge engineer'' interviews experts in a certain domain and tries to embody their knowledge in a computer program for carrying out some task. How well this works depends on whether the intellectual mechanisms required for the task are within the present state of AI. When this turned out not to be so, there were many disappointing results. One of the first expert systems was MYCIN in 1974, which diagnosed bacterial infections of the blood and suggested treatments. It did better than medical students or practicing doctors, provided its limitations were observed. Namely, its ontology included bacteria, symptoms, and treatments and did not include patients, doctors, hospitals, death, recovery, and events occurring in time. Its interactions depended on a single patient being considered. Since the experts consulted by the knowledge engineers knew about patients, doctors, death, recovery, etc., it is clear that the knowledge engineers forced what the experts told them into a predetermined framework. In the present state of AI, this has to be true. The usefulness of current expert systems depends on their users having common sense.

6. Heuristic classification

One of the most feasible kinds of expert system given the present knowledge of AI is to put some information in one of a fixed set of categories using several sources of information. An example is advising whether to accept a proposed credit card purchase. Information is available about the owner of the credit card, his record of payment and also about the item he is buying and about the establishment from which he is buying it (e.g., about whether there have been previous credit card frauds at this establishment).

7. Commonsense reasoning

It is the branch of Artificial intelligence concerned with replicating human thinking. In theory, if the computer is endowed with good Knowledge Representation Database, including a comprehensive common sense database, and is able to process and

respond in plain-text English, it will have the ability to process and reason with English texts. The task of Common Sense Knowledge is probably the least studied area, though it is included in many ways in knowledge representation task. There are two issues with this .one is how to represent the knowledge gathered in a computer processible, and human accessible way. The second task is actually collecting the Common Sense knowledge. There are a couple of different groups who are doing this now. Knowledge Gathering is usually done for expert systems and is limited in its breadth to a limited domain. The two common sense projects are Open Mind Common Sense and Cycorp. To investigate this sort of problems General Problem Solver was developed.

Problem and Problem Spaces

State space search is a powerful technique for solving problems. However, in order to apply this technique we need to formulate the problem. A problem can be defined formally by 4 components:

- **Initial state** :- Specify one or more states within that space that describe possible situations from which the problem solving process may start. These states are called the initial sates.
- **Successor function S(x)** :- Description of all possible actions available. Given a particular state x, S(x) returns a set of < action, successor> ordered pairs in which each action is one of the legal actions in state x and each successor is a state that can be reached from x by applying the action.
- **Goal Test** :– Goal states are one or more states that would be acceptable as solutions to the problem. Goal test determines whether a given state is a goal state. They may or may not be explicitly specified.
- **Path cost** :- Function that assigns a numeric cost to a path. It is usually the sum of the costs of individual actions along the path. Step cost of taking action ‘a’ to go from state ‘x’ to ‘y’ is denoted by c(x,a,y).

Problem Space is the environment in which search takes place. **Problem Instance** is the problem space combined with initial and goal state.

State space is the set of all states reachable from the initial state.(initial state + successor function).State space forms a graph in which nodes are states and the arcs between nodes are actions. A **path** in state space is a sequence of states connected by a sequence of actions.

A **solution** is a sequence of actions (path) leading from the initial state to a goal state. Solution quality is measured by the path cost function. An **Optimal solution** has the lowest path cost among all solutions.

Example Problems

1. 8 Puzzle problem

The eight puzzle consists of a three by three board with eight numbered tiles and a blank space. A tile adjacent to the blank space can slide into the space. A game consists of a starting position and a specified goal position. The goal is to transform the starting position into the goal position by sliding the tiles around. The object is to figure out the steps needed to get from one configuration of the tiles to another.

Arrange the tiles so that all the tiles are in the correct positions. You do this by moving tiles. You can move a tile up, down, left, or right, so long as the following conditions are met:
A) there's no other tile blocking you in the direction of the movement; and
B) you're not trying to move outside of the boundaries/edges.

The **8-puzzle** - also known as the **sliding-block/tile-puzzle** - is one of the most popular instrument in the artificial intelligence (AI) studies. It belongs to AI exercises commonly referred as **toy-problems**.

For example, suppose we wanted to get from the initial state to the goal state given below.

Initial state: Goal state:

$$\begin{bmatrix} 2 & 8 & 3 \\ 1 & 6 & 4 \\ 7 & \square & 5 \end{bmatrix} \qquad \begin{bmatrix} 1 & 2 & 3 \\ 8 & \square & 4 \\ 7 & 6 & 5 \end{bmatrix}$$

and the $\square$ is a blank tile you can slide around. This 8 puzzle instance can be solved in 5 steps.

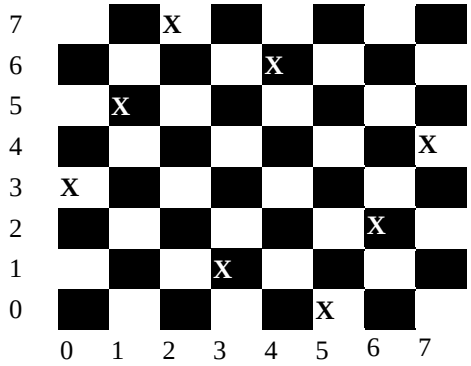
1. **States** – A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
2. **Initial state** – Any state can be designated as the initial state. Any given goal can be reached from exactly half of the possible initial states.
3. **Successor function**. – generates the legal states that result from trying the four actions(blank moves left, right, up, down).
4. **Goal test** – this checks whether the state matches the goal configuration.
5. **Path cost** - Number of actions to reach goal.

2. 8-Queens Problem

The goal of 8-queens problem is to place 8 queens on a 8 by 8 chess board so that no queen can attack any other queen. A queen attacks another if they are in the same row either vertically, horizontally, or diagonally.

If we want to find a single solution, it is not difficult. If we want to find all possible solutions, the problem is difficult and backtrack method is the only known method. For 8-queen, we have 92 solutions. If we exclude symmetry, there are 12 solutions.

One Solution to the 8-Queens Problem:



Although efficient special purpose algorithms exist for this problem and the whole 'n' queens family, it remains an interesting test problem for search algorithms. There are 2 main kinds of formulation.

1.**Incremental formulation** – involves operators that augment the state description starting with an empty state , ie each action adds a queen to the board.
2.**Complete-state formulation** – starts with all 8 queens on the board and moves them abroad.

In either case the path cost is of no interest because only the final state counts.
For eg: Incremental formulation

1. **States** - Any arrangement of 0 to 8 queens on the board
2. **Initial state** - No queens on the board
3. **Successor func**tion - Add a queen to any empty square
4. **Goal test** - 8 queens on board and none attacked

There are 3×10^{14} possible sequences to investigate

3. Water-Jug Problem

A Water Jug Problem: You are given two jugs, a 4-gallon one and a 3-gallon one. 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 2 gallons of water into the 4-gallon jug?

The state space for this problem can be described as the set of ordered pairs of integers (x, y), such that $x = 0, 1, 2, 3,$ or 4 and $y = 0, 1, 2,$ or 3; x represents the number of gallons of water in the 4-gallon jug, and y represents the quantity of water in the 3-gallon jug. The start state is (0, 0). The goal state is $(2, n)$ for any value of n (since the problem does not specify how many gallons need to be in the 3-gallon jug).

The operators[2] to be used to solve the problem can be described as shown in Fig. 2.3. As in the chess problem, 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. Notice that in order to describe the operators completely, it was necessary to make explicit some assumptions not mentioned in the problem statement. We have assumed that we can fill a jug from the pump, that we can pour water out of a jug onto the ground, that we can pour water from one jug to another, and that 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.

To solve the water jug problem, all we need, in addition to the problem description given above, is a control structure that loops through a simple cycle in which some rule whose left side matches the current state is chosen, the appropriate change to the state is made as described in the corresponding right side, and the resulting state is checked to see if it corresponds to a goal state. As long as it does not, the cycle continues. Clearly the speed with which the problem gets solved depends on the mechanism that is used to select the next operation to be performed. In Chapter 3, we discuss several ways of making that selection.

For the water jug problem, as with many others, there are several sequences of operators that solve the problem. One such sequence is shown in Fig. 2.4. Often, a problem contains the explicit or implied statement that the shortest (or cheapest) such sequence be found. If present, this requirement will have a significant effect on the choice of an appropriate mechanism to guide the search for a solution. We discuss this issue in Section 2.3.4.

Several issues that often arise in converting an informal problem statement into a formal problem description are illustrated by this sample water jug problem. The first of these issues concerns the role of the conditions that occur in the left sides of the rules. All but one of the rules shown in Fig. 2.3 contain conditions that must be satisfied before the operator described by the rule can be applied. For example, the first rule says, "If the 4-gallon jug is not already full, fill it." This rule could, however, have been written as, "Fill the 4-gallon jug," since it is physically possible to fill the jug even if it is already full. It is stupid to do so since no change in the problem state results, but it is possible. By encoding in the left sides of the rules constraints that are not strictly necessary but that restrict the application of the rules to states in which the rules are most likely to lead to a solution, we can generally increase the efficiency of the problem-solving program that uses the rules.

1	(x, y) if $x < 4$	→	$(4, y)$	Fill the 4-gallon jug
2	(x, y) if $y < 3$	→	$(x, 3)$	Fill the 3-gallon jug
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 $x + y \geq 4$ and $y > 0$	→	$(4, y - (4 - x))$	Pour water from the 3-gallon jug into the 4-gallon jug until the 4-gallon jug is full
8	(x, y) if $x + y \geq 3$ and $x > 0$	→	$(x - (3 - y), 3)$	Pour water from the 4-gallon jug into the 3-gallon jug until the 3-gallon jug is full
9	(x, y) if $x + y \leq 4$ and $y > 0$	→	$(x + y, 0)$	Pour all the water from the 3-gallon jug into the 4-gallon jug
10	(x, y) if $x + y \leq 3$ and $x > 0$	→	$(0, x + y)$	Pour all the water from the 4-gallon jug into the 3-gallon jug
11	$(0, 2)$	→	$(2, 0)$	Pour the 2 gallons from the 3-gallon jug into the 4-gallon jug
12	$(2, y)$	→	$(0, y)$	Empty the 2 gallons in the 4-gallon jug on the ground

Fig. 2.3 *Production Rules for the Water Jug Problem*

Gallons in the 4-Gallon Jug	Gallons in the 3-Gallon Jug	Rule Applied
0	0	
		2
0	3	
		9
3	0	
		2
3	3	
		7
4	2	
		5 or 12
0	2	
		9 or 11
2	0	

You are given two jugs a 4-gallon one and other 3-gallon one. Neither has any measuring marker on it. There is a pump that can be used to fill the jugs with water. The Problem : How can you get exactly 2 gallons of water into the 4-gallon jug?

The State Space for this problem can be described as a set of Ordered pairs of integers (x, y) such that x = 0, 1, 2, 3, or 4 and y = 0, 1, 2 , or 3; 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.

The Start State is (0, 0).The Goal States are (2, 0), (2, 1), (2, 2), (2, 3).The actions include :

1) fill either jug with water
2) empty either jug
3) pour water from 4-gallon jug to 3-gallon jug
4) pour water from 3-gallon jug to 4-gallon jug

4. Towers of Hanoi

There are three posts 1, 2, and 3. One of which, say post 1, has ‘n’ disks on it in ascending size. The Problem is: How to transfer all disks from the post 1 to post 2,

keeping the order of ascending size. Each time only one disk can be moved. Post 2 is used for the intermediate storage

A good example of a problem solving domain is the "Tower of Hanoi" puzzle. This puzzle involves a set of three rings of different sizes that can be placed on three different pegs, like so:

The goal of this puzzle is start with the rings arranged as shown above, and to move them all to this configuration:

Only the top ring on a peg can be moved, and it may only be placed on a smaller ring, or on an empty peg. For the Tower of Hanoi, the problem space is the set of all possible configurations of rings on the pegs. Some of the configurations or states in the space are given special interpretations:

Initial States

States where a given episode of problem solving starts. In the Tower of Hanoi puzzle, there is one initial state, as shown above. In other domains there may be a number of possible initial states.

Goal or Solution States

States that are considered solutions to the problem. Again, the Tower of Hanoi problem has a single solution state, but usually there are a number of states that count as solutions.

Failure or Impossible States

In some domains, there are states with the property that if they are ever encountered, the problem solving is considered a failure. In the Tower of Hanoi domain, one might define failure states as any in which the rule that rings can only be placed on bigger rings is violated

Real World Problems

1. Route finding problem

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

such as routing in computer networks, military operations planning and airline travel planning system. These problems are typically complex to explain.

2. Travelling salesman problem

This is also a Route finding problem. A salesperson has to visit a number of cities. (S)He can start at any city and must finish at that same city. The salesperson must visit each city only once. A simple control structure could in principle solve the problem. It would simply explore all possible paths in the tree and return the one with the shortest length. This approach will work for very short list of cities. But it breaks down quickly as the no of cities grows.

If there are n cities then the number of possible paths among them is 1*2*3*4……*(n-1) or (n-1)! .The time required to examine a single path is proportional to n!. Assume there are 35 cities to visit. To solve this problem he would take more time than he would be willing to send. This phenomenon is called **combinatorial explosion.**

We can beat this using a simple strategy 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. Although this algorithm is efficient than the first one it still requires exponential time.

3. VLSI layout

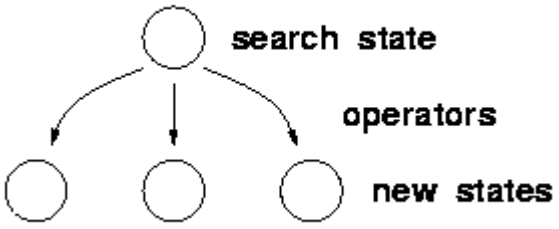
A VLSI layout problem requires positioning millions of components and connections on a chip to minimize area, minimize circuit delay and maximize manufacturing yield. The layout problem comes after the logical design phase and is usually split into 2 parts : cell layout and channel routing. The aim is to place the cells on the chip so that they do not overlap and so that there is room for the connecting wires to be placed between the cells. Channel routing finds a specific route for each wire through the gaps between the cells.

4. Robot Navigation

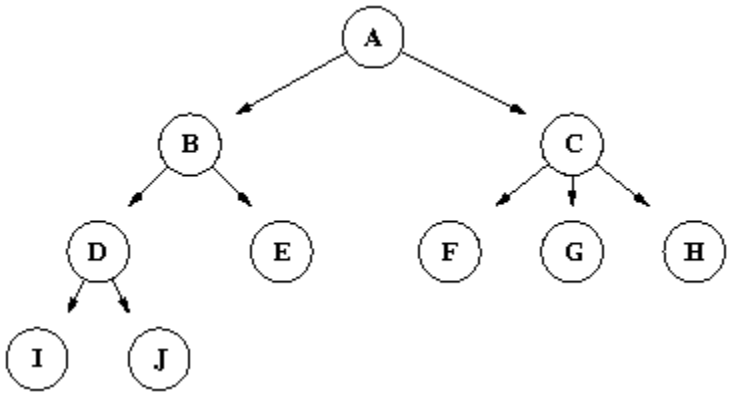
It is a generalization of route problem. Rather than a discrete set of routes, a robot can move in a continuous space with an infinite set of possible actions and states. When robot has arms and legs or wheel that must also be controlled, the search space becomes multi-dimensional.

Problem solving by Searching

Search is a classical AI topic. Many AI problems reduce to searching for solutions in the problem state space. A "search space" is pretty much the same as a problem space, but the term is used specifically when a search algorithm is used to solve the problem. Search spaces are often drawn by showing the individual states, with the states that result from applying the domain's operators below them, like this:



We can now draw a part of a search space by showing the initial state at the top of an upside-down tree, with each "level" consisting of states that result from applying operators to states in the level above:



Problem solving agents decide what to do by finding sequences of actions that lead to desirable states. **Search algorithms** classified as

1. **Uninformed Search/Blind Search** - in the sense that they are given no information about the problem other than its problem definition.

2. **Informed Search/Heuristic Search –** ones that have some idea of where to look for solutions.

Goals help organize behavior by limiting the objectives that the agent is trying to achieve. Goal formulation based on the current situation and the agent's performance measure is the first step in problem solving. The agent's task is to find out which sequence of actions will get it to a goal state. **Problem formulation** is the process of deciding what actions and states to consider given a goal.

An agent 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. The process of looking for such a sequence is called search. A search algorithm takes a problem as input and returns a solution in the form of an action sequence. Once a solution is found the actions it recommends can be carried out. This is called **execution phase**. Thus we have a simple **formulate, search, execute** design for an agent.

Usually the environment is assumed to be static because formulating and solving the problem is done without paying attention to any changes that might be occurring in the environment.

Searching for Solutions

Solution can be found out by searching in state space. We can generate a **search tree** that is generated by the initial state and the successor function that together define the state space. The root of search tree is a **search node** corresponding to the initial state. The first step is to find whether it is a goal state. If it is not we need to consider other states. This is done by **expanding** the current state ie applying the successor function to the current state thereby **generating** a new set of states.

We continue choosing, testing and expanding until either a solution is found or there are no more states to be expanded. The choice of which state to expand is determined by **search strategy**. For e.g. if there are 20 states in state space, one for each city. But there are infinite no of paths in this state space so search tree has infinite no of nodes.

Nodes are usually represented by a data structure with 5 components as:

1. **State** – the state in the state space to which the node corresponds
2. **Parent node** – node in the search tree that generated this node
3. **Action** – action that was applied to the parent to generate the node.
4. **Path cost** – cost of the path from the initial state to the node as indicated by parent pointers
5. **Depth** – no of steps along the path from the initial state.

Collection of nodes that have been generated but not yet expanded is called **fringe**. Each element of the fringe is a **leaf node** i.e. a node with no successors in the tree.

Measuring Problem-solving performance

Problem-solving performance is measured in four ways:

1. **Completeness** - Does it always find a solution if one exists?
2. **Optimality** - Does it always find the least-cost solution?
3. **Time Complexity** - How long does it take to find a solution? (Number of nodes generated/expanded)

4. **Space Complexity** - Number of nodes stored in memory during search?

Time and space complexity are measured in terms of problem difficulty defined by:

b - Maximum branching factor of the search tree
d - Depth of the least-cost solution
m - Maximum depth of the state space (may be infinity)

Time is often measured in terms of no of nodes generated during search and space in terms of max no of nodes stored in memory.

2.2 PRODUCTION SYSTEMS

Since search forms the core of many intelligent processes, it is useful to structure AI programs in a way that facilitates describing and performing the search process. Production systems provide such structures. A definition of a production system is given below. Do not be confused by other uses of the word *production*, such as to describe what is done in factories. A *production system* consists of:

- A set of rules, each consisting of a left side (a pattern) that determines the applicability of the rule and a right side that describes the operation to be performed if the rule is applied.[3]
- One or more knowledge/databases that contain whatever information is appropri- ate for the particular task. Some parts 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.
- 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.
- A rule applier.

So far, our definition of a production system has been very general. It encompasses a great many systems, including our descriptions of both a chess player and a water jug problem solver. It also encompasses a family of general production system interpreters, including:

- Basic production system languages, such as OPS5 [Brownston *et al.*, 1985] and ACT* [Anderson, 1983].
- More complex, often hybrid systems called *expert system shells*, which provide complete (relatively speaking) environments for the construction of knowledge- based expert systems.
- General problem-solving architectures like SOAR [Laird *et al.*, 1987], a system based on a specific set of cognitively motivated hypotheses about the nature of problem-solving.

All of these systems provide the overall architecture of a production system and allow the programmer to write rules that define particular problems to be solved. We discuss production system issues further in Chapter 6.

We have now seen that in order to solve a problem, we must first reduce it to one for which a precise statement can be given. This can be done by defining the problem's state space (including the start and goal states) and a set of operators for moving in that space. The problem can then be solved by searching for a path through the space from an initial state to a goal state. The process of solving the problem can usefully be modeled as a production system. In the rest of this section, we look at the problem of choosing the appropriate control structure for the production system so that the search can be as efficient as possible.

2.2.1 Control Strategies

So far, we have completely ignored the question of how to decide which rule to apply next during the process of searching for a solution to a 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.

- *The first requirement of a good control strategy is that it causes motion.* Consider again the water jug problem of the last section. 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.
- *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. Fig. 2.5 shows how the tree looks at this point. Now for each leaf node, generate all its successors by applying all the rules that are appropriate. The tree at this point is shown in Fig. 2.6.[4] Continue this process until some rule produces a goal state. This process, called *breadth-first search*, can be described precisely as follows.

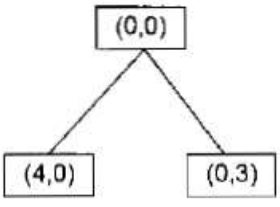


Fig. 2.5 *One Level of a Breadth-First Search Tree*

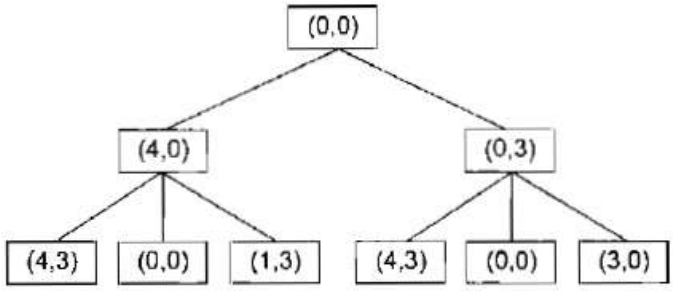


Fig. 2.6 *Two Levels of a Breadth-First Search Tree*

Algorithm: Breadth-First Search

1. Create a variable called *NODE-LIST* and set it to the initial state.
2. Until a goal state is found or *NODE-LIST* is empty:
   (a) Remove the first element from *NODE-LIST* and call it *E*. If *NODE-LIST* was empty, quit.
   (b) For each way that each rule can match the state described in *E* do:
      (i) Apply the rule to generate a new state,
      (ii) If the new state is a goal state, quit and return this state.
      (iii) Otherwise, add the new state to the end of *NODE-LIST*.

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 prespecified "futility" limit. In such a case, backtracking occurs. The most recently created state from which alternative 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 form of backtracking is what is usually meant by the simple term *backtracking*. But there are other ways of retracting steps of a computation. We discuss one important such way, dependency-directed backtracking, in Chapter 7. Until then, though, when we use the term backtracking, it means chronological backtracking.

The search procedure we have just described is also called *depth-first search*. The following algorithm describes this precisely.

Algorithm: Depth-First Search

1. If the initial state is a goal state, quit and return success.
2. Otherwise, do the following until success or failure is signaled:
   (a) Generate a successor, *E*, of the initial state. If there are no more successors, signal failure.
   (b) Call Depth-First Search with *E* as the initial state.
   (c) If success is returned, signal success. Otherwise continue in this loop.

Figure 2.7 shows a snapshot of a depth-first search for the water jug problem. A comparison of these two simple methods produces the following observations:

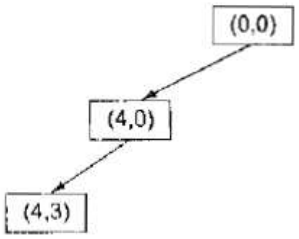


Fig. 2.7 *A Depth-First Search Tree*

Advantages of Depth-First Search

- Depth-first search requires less memory since only the nodes on the current path are stored. This contrasts with breadth-first search, where all of the tree that has so far been generated must be stored.
- By chance (or if care is taken in ordering the alternative successor states), depth-first search may find a solution without examining much of the search space at all. This contrasts with breadth-first search in which all parts of the tree must be examined to level n before any nodes on level $n + 1$ can be examined. This is particularly significant if many acceptable solutions exist. Depth-first search can stop when one of them is found.

Advantages of Breadth-First Search

- Breadth-first search will not get trapped exploring a blind alley. This contrasts with depth-first searching, which may follow a single, unfruitful path for a very long time, perhaps forever, before the path actually terminates in a state that has no successors. This is a particular problem in depth-first search if there are loops (i.e., a state has a successor that is also one of its ancestors) unless special care is expended to test for such a situation. The example in Fig. 2.7, if it continues always choosing the first (in numerical sequence) rule that applies, will have exactly this problem.
- 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. This contrasts with depth-first search, which may find a long path to a solution in one part of the tree, when a shorter path exists in some other, unexplored part of the tree.

Clearly what we would like is a way to combine the advantages of both of these methods. In Section 3.3 we will talk about one way of doing this when we have some additional information. Later, in Section 12.5, we will describe an uninformed way of doing so.

For the water jug problem, most control strategies that cause motion and are systematic will lead to an answer. The problem is simple. But this is not always the case. In order to solve some problems during our lifetime, we must also demand a control structure that is efficient.

Consider the following problem.

> **The Traveling Salesman Problem:** A salesman has a list of cities, each of which he must visit exactly once. There are direct roads between each pair of cities on the list. Find the route the salesman should follow for the shortest possible round trip that both starts and finishes at any one of the cities.

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 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 $1.2...(N-1)$, 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 3,628,800, which is a very large number. The salesman could easily have 25 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.

2.2.2 Heuristic Search

In order to solve many hard problems efficiently, it is often necessary to compromise the requirements of mobility and systematicity and to construct a control structure that is no longer guaranteed to find the best answer but that will almost always find a very good answer. Thus we introduce the idea of a heuristic.[5] A

heuristic is a technique that improves the efficiency of a search process, possibly by sacrificing claims of completeness. Heuristics are like tour guides. They are good to the extent that they point in generally interesting directions; they are bad to the extent that they may miss points of interest to particular individuals. Some heuristics help to guide a search process without sacrificing any claims to completeness that the process might previously have had. Others (in fact, many of the best ones) may occasionally cause an excellent path to be overlooked. But, on an average, they improve the quality of the paths that are explored. Using good heuristics, we can hope to get good (though possibly nonoptimal) solutions to hard problems, such as the traveling salesman, in less than exponential time. There are some good general-purpose heuristics that are useful in a wide variety of problem domains. In addition, it is possible to construct special-purpose heuristics that exploit domain-specific knowledge to solve particular problems.

One example of a good general-purpose heuristic that is useful for a variety of combinatorial problems is the *nearest neighbor heuristic*, which works by selecting the locally superior alternative at each step. 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 2 until all cities have been visited.

This procedure executes in time proportional to N^2, a significant improvement over $N!$, and it is possible to prove an upper bound on the error it incurs. For general-purpose heuristics, such as nearest neighbor, it is often possible to prove such error bounds, which provides reassurance that one is not paying too high a price in accuracy for speed.

In many AI problems, however, it is not possible to produce such reassuring bounds. This is true for two reasons:

- For real world problems, it is often hard to measure precisely the value of a particular solution. Although the length of a trip to several cities is a precise notion, the appropriateness of a particular response to such questions as "Why has inflation increased?" is much less so.
- For real world problems, it is often useful to introduce heuristics based on relatively unstructured knowledge. It is often impossible to define this knowledge in such a way that a mathematical analysis of its effect on the search process can be performed.

There are many heuristics that, although they are not as general as the nearest neighbor heuristic, are nevertheless useful in a wide variety of domains. For example, consider the task of discovering interesting ideas in some specified area. The following heuristic [Lenat, 1983b] is often useful:

If there is an interesting function of two arguments $f(x, y)$, look at what happens if the two arguments are identical.

In the domain of mathematics, this heuristic leads to the discovery of *squaring iff* is the multiplication function, and it leads to the discovery of an *identity* function if f is the function of set union. In less formal domains, this same heuristic leads to the discovery of *introspection* if f is the function contemplate or it leads to the notion of *suicide iff* is the function kill.

Without heuristics, we would become hopelessly ensnarled in a combinatorial ex-plosion. This alone might be a sufficient argument in favor of their use. But there are other arguments as well:

- Rarely do we actually need the optimum solution; a good approximation will usually serve very well. In fact, there is some evidence that people, when they solve problems, are not optimizers but rather are *satisficers* [Simon, 1981]. In other words, they seek any solution that satisfies some set of requirements, and as soon as they find one they quit. A good example of this is the search for a parking space. Most people stop as soon as they find a fairly good space, even if there might be a slightly better space up ahead.

- Although the approximations produced by heuristics may not be very good in the worst case, worst cases rarely arise in the real world. For example, although many graphs are not separable (or even nearly so) and thus cannot be considered as a set of small problems rather than one large one, a lot of graphs describing the real world are.[6]
- Trying to understand why a heuristic works, or why it doesn't work, often leads to a deeper understanding of the problem.

One of the best descriptions of the importance of heuristics in solving interesting problems is *How to Solve It* [Polya, 1957]. Although the focus of the book is the solution of mathematical problems, many of the techniques it describes are more generally applicable. For example, given a problem to solve, look for a similar problem you have solved before. Ask whether you can use either the solution of that problem or the method that was used to obtain the solution to help solve the new problem. Polya's work serves as an excellent guide for people who want to become better problem solvers. Unfortunately, it is not a panacea for AI for a couple of reasons. One is that it relies on human abilities that we must first understand well enough to build into a program. For example, many of the problems Polya discusses are geometric ones in which once an appropriate picture is drawn, the answer can be seen immediately. But to exploit such techniques in programs, we must develop a good way of representing and manipulating descriptions of those Fig.s. Another is that the rules are very general.

They have extremely underspecified left sides, so it is hard to use them to guide a search—too many of them are applicable at once. Many of the rules are really only useful for looking back and rationalizing a solution after it has been found. In essence, the problem is that Polya's rules have not been operationalized.

Nevertheless, Polya was several steps ahead of AI. A comment he made in the preface to the first printing (1944) of the book is interesting in this respect:

The following pages are written somewhat concisely, but as simply as possible, and are based on a long and serious study of methods of solution. This sort of study, called *heuristic* by some writers, is not in fashion nowadays but has a long past and, perhaps, some future.

There are two major ways in which domain-specific, heuristic knowledge can be incorporated into a rule-based search procedure:

- In the rules themselves. For example, the rules for a chess-playing system might describe not simply the set of legal moves but rather a set of "sensible" moves, as determined by the rule writer.
- As a heuristic function that evaluates individual problem states and determines how desirable they are.

A *heuristic function* is a function that maps from problem state descriptions to measures of desirability, usually represented as numbers. Which aspects of the problem state are considered, how those aspects are evaluated, and the weights given to individual aspects are chosen in such a way that the value of the heuristic function at a given node in the search process gives as good an estimate as possible of whether that node is on the desired path to a solution.

Well-designed heuristic functions can play an important part in efficiently guiding a search process toward a solution. Sometimes very simple heuristic functions can provide a fairly good estimate of whether a path is any good or not. In other situations, more complex heuristic functions should be employed. Fig. 2.8 shows some simple heuristic functions for a few problems. Notice that sometimes a high value of the heuristic function indicates a relatively good position (as shown for chess and tic-tac- toe), while at other times a low value indicates an advantageous situation (as shown for the traveling salesman). It does not matter, in general, which way the function is stated. The program that uses the values of the function can attempt to minimize it or to maximize it as appropriate.

Production Systems

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 a basic representation found 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.

Productions consist 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*. A production system also contains a database, sometimes called working memory, which maintains data about current state or knowledge, and a rule interpreter. The rule interpreter must provide a mechanism for prioritizing productions when more than one is triggered.

Production systems consist of a **database of rules, a working memory, a matcher, and a procedure** that resolves conflicts between rules.

Matching

The rules of a production consist of a condition and action in the form: (**if** *x* **then** *y*). The left-hand-side conditions (*x* and *y* may be arbitrarily complex conjunctions of expressions) are compared against the elements of working memory to determine if the conditions are satisfied.

Conflict Resolution

At any point in processing, several productions may match to the elements of working memory simultaneously. Since production systems are normally implemented on serial computers, this result in a *conflict*: there is a non-unique choice about which action to take next. 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.

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.

Problem Characteristics

To build a system to solve a particular problem we need to do 4 things.

1) Define the problem precisely. The definition must include precise specifications of what the initial situations will be as well as what final situations constitute acceptable solutions to the problem.

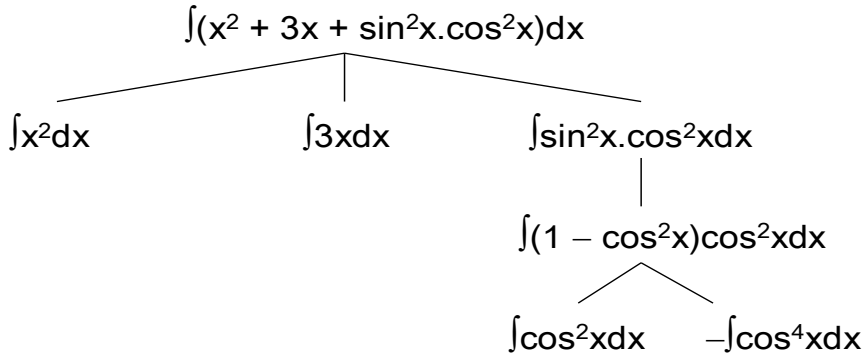
2) Analyse the problem

3) Isolate and represent the task knowledge that is necessary to solve the problem.

4) Choose the best problem solving technique and apply it to the particular problem.

In order to choose an appropriate method for a particular problem it is necessary to analyze the problem along several key dimensions as:

- Is the problem decomposable?
- Can solution steps be ignored or undone?
- Is the universe predictable?
- Is a good solution absolute or relative?
- Is the solution a state or a path?
- What is the role of knowledge?
- Does the task require human-interaction?



1. Is the problem decomposable?

Suppose we want to solve the problem of computing the expression:

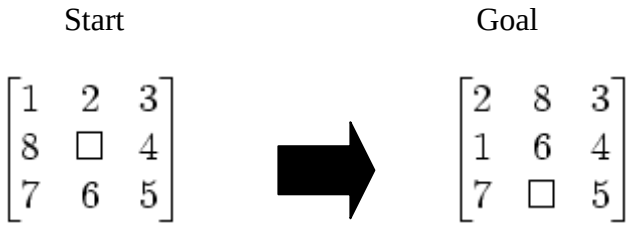
∫(x2 + 3x + sin2x.cos2x)dx

We can solve this problem by breaking down it into 3 smaller problems each of which we can be solved independently. Decomposable problem can be solved easily. This can be solved by a simple integration program that works as follows. At each step it checks to see whether the problem it is working on is immediately solvable. If so then the answer is returned directly. If the problem is not easily solvable, the integrator checks to see whether it can decompose the problem into smaller problems. If it can it creates those problems and calls itself recursively on them. The problem tree is shown below:

2. Can solution steps be ignored or undone?

In attempting to solve the 8-Puzzle we might make a stupid move. For eg in the game shown above we might start by sliding tile 5 into the empty space. Having done that we cannot change our mind and immediately slide tile 6 into the empty space since the empty space will essentially have moved. But we can backtrack and undo the first move sliding tile 5 back to where it was. Then we can move tile 6.

An additional step must be performed to undo each incorrect step, whereas no action was required to undo a useless lemma. In addition the control mechanism for an 8-Puzzle solver must keep track of the order in which operations are performed so that the operations can be undone one at a time if necessary.



Problem Classification

1) **Ignorable problems** - in which solution steps can be ignored. For eg in the case of Theorem Proving a lemma that has been proved can be ignored for next steps. Ignorable problems can be solved using a simple control structure that never backtracks. Such a control structure is easy to implement.

2) **Recoverable problems** - in which solution steps can be undone and backtracked. Eg :8- Puzzle. Recoverable problems can be solved by a slightly more complicated control strategy that does sometimes make mistakes. Backtracking will be necessary to recover from such mistakes, so the control structure must be implemented using a pushdown stack in which decisions are recorded in case they need to be undone.

3) **Irrecoverable problems** - in which moves cannot be retracted. Eg : Playing Chess. They will need to be solved by a system that expends a great deal of effort making each decision since the decision must be final. Irrecoverable problems can be solved by recoverable style methods via planning.

3. Is the universe predictable?

In 8-Puzzle it is possible to plan an entire sequence of moves and be confident that we know what the resulting state will be. These are called **certain outcome problems**. For certain-outcome problems, planning can used to generate a sequence of operators that is guaranteed to lead to a solution.

In Bridge we cannot know exactly where all the cards are or what the other players will do on their turns. What we would like to do is to plan the entire hand before

making the first play. These problems are called uncertain outcome problems. For **uncertain-outcome problems**, a sequence of generated operators can only have a good probability of leading to a solution.

Plan revision is made as the plan is carried out and the necessary feedback is provided. One of the hardest types of problems to solve is the irrecoverable uncertain outcome. Eg : controlling a robot arm.

4. Is a good solution absolute or relative?

Different reasoning paths lead to the answer. It does not matter which path we follow. For eg in the case of travelling Salesman Problem we have to try all paths to find the shortest one. **Any-path problems** can be solved in a reasonable amount of time using heuristics that suggest good paths to explore. If the heuristics are not perfect the search for the solution may not be as directly as possible. For **best-path problems**, much more exhaustive search will be performed. Best –path problems are computationally harder than any – path problems.

But now consider again the traveling salesman problem. Our goal is to find the shortest route that visits each city exactly once. Suppose the cities to be visited and the distances between them are as shown in Fig. 2.14.

	Boston	New York	Miami	Dallas	S.F.
Boston		250	1450	1700	3000
New York	250		1200	1500	2900
Miami	1450	1200		1600	3300
Dallas	1700	1500	1600		1700
S.F.	3000	2900	3300	1700	

One place the salesman could start is Boston. In that case, one path that might be followed is the one shown in Fig. 2.15, which is 8850 miles long. But is this the solution to the problem? The answer is that we cannot be sure unless we also try all other paths to make sure that none of them is shorter. In this case, as can be seen from Fig. 2.16, the first path is definitely not the solution to the salesman's problem.

These two examples illustrate the difference between any-path problems and best- path problems. Best-path problems are, in general, computationally harder than any-path problems. Any-path problems can often be solved in a reasonable amount of time by using heuristics that suggest good paths to explore. (See the discussion of best-first search in Chapter 3 for one way of doing this.) If the heuristics are not perfect, the search for a solution may not be as direct as possible, but that does not matter. For true best-path problems, however, no heuristic that could possibly miss the best solution can be used. So a much more exhaustive search will be performed.

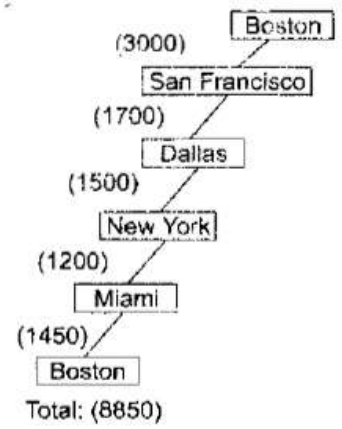


Fig. 2.15 *One Path among the Cities*

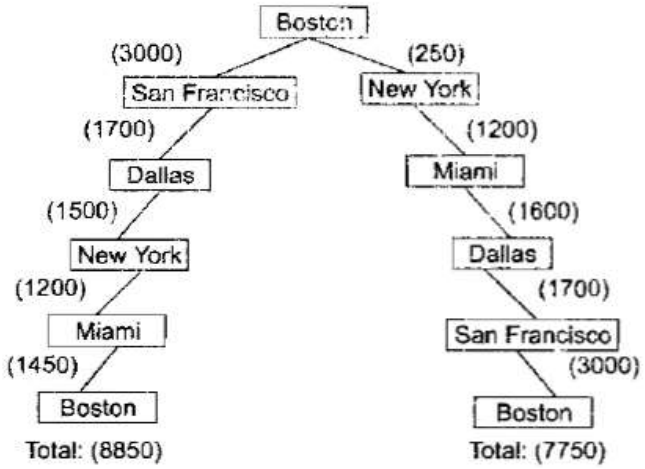


Fig. 2.16 *Two Paths Among the Cities*

5. Is the solution a state or a path?

In the case of water Jug Problem solution is a path to the state. The path that leads to the goal must be reported. But in the case of natural language understanding solution is a state in the world. At one level this difference can be ignored and all problems can be formulated as ones in which only a state is required to be reported. A path-solution problem can be reformulated as a state-solution problem by describing a state as a partial path to a solution. The question is whether that is natural or not.

6. What is the role of knowledge?

Consider the problem of playing chess. Suppose you had unlimited computing power available. How much knowledge would be required by a perfect program? It requires just the rules for determining legal moves and some simple control mechanism that implements an appropriate search procedure. In this case knowledge is important only to constrain the search for a solution.

But now consider the problem of scanning daily newspaper to decide who will win in the upcoming elections. Suppose you had unlimited computing power available .But in this case lot of knowledge is required even to be able to recognize a solution.

7. Does the task require human-interaction?

Based on human interaction problems can be classified as:

- **Solitary problem**, in which computer is given a problem description and produces an answer with no intermediate communication and no demand for an explanation of the reasoning process.

- **Conversational problem**, in which there is intermediate communication between a person and a computer either to provide additional assistance to the computer or to provide additional information to the user or both.

For eg while proving mathematical theorem it should be made sure that the proof exists. If there is that much difficult it may not know where to start.

Search Strategies

1. Exhaustive (uninformed) search

- Breath-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search
- Bidirectional search

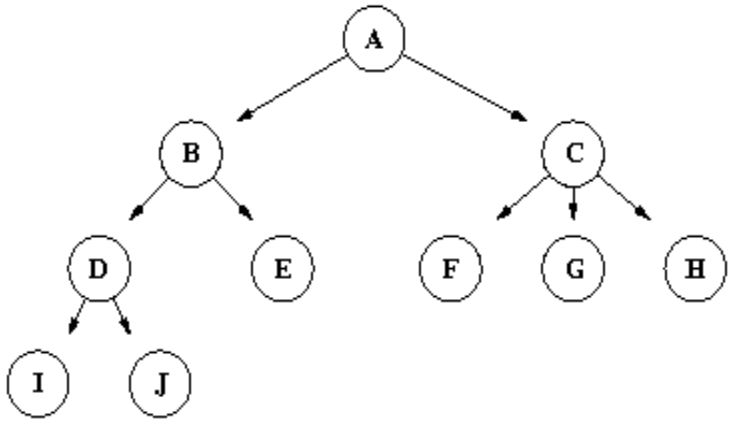
2. Heuristic (informed) search

- Hill climbing
- Simulated annealing
- Best-first search
- A* search

Uninformed/Blind Search

1. Breadth First Search

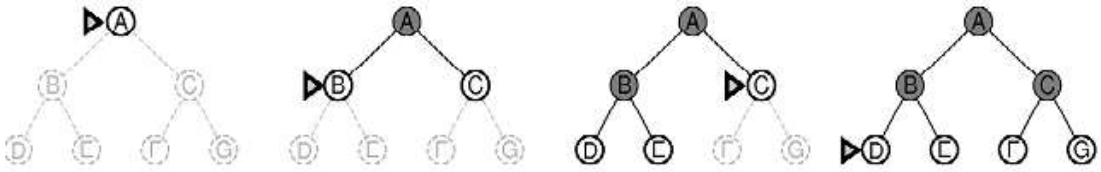
It is a simple strategy in which root node is expanded first then all successors of root node are expanded next and then their successors and so on. Consider again the search space shown below:



Let us suppose that the state labeled G is a goal state in this space, and that, as shown, no operators apply to the states I, J, E, F and H.

A program will start with the initial state A, and try to find the goal by applying operators to that state, and then to the states B and/or C that result, and so forth. The idea is to find a goal state, and often one is also interested in finding the goal state as fast as possible, or in finding a solution that requires the minimum number of steps, or satisfies some other requirements.

One approach to deciding which states that operators will be applied to is called "Breadth-First Search". In a breadth-first search, all of the states at one level of the tree are considered before any of the states at the next lower level. So for the simple search state pictured above, the states would be applied in the order as indicated below:



Algorithm:

1. Create a variable called NODE-LIST and set it to initial state.
2. Until a goal is found or NODE-LIST is non-empty,
    1. Remove the first element from NODE-LIST and call it E. If NODE-LIST was empty, quit.
    2. For each way that each rule can match the state described in E do
        a) Apply the rule to generate a new state.
        b) If the new state is a goal state, terminate with success and return that state.
        c) Otherwise add the new state to the end of NODE-LIST

Usually BFS is implemented using data structure called *queue*. You add the newly formed paths to the back of the list. When you remove them to expand them, you remove them from the front.

Completeness: Does it always find a solution if one exists? YES, if shallowest goal node is at some finite depth d

Time complexity: Assume a state space where every state has b successors. Root node has b successors, each node at the next level has again b successors (total b^2), and so on. Assume solution is at depth d. In Worst case we would expand all but the last node at depth d since the goal itself is not expanded generating b^{d+1} –b nodes at level d+1.Then total number of nodes generated is

$b+b^2+b^3+...+b^d+(b^{d+1}-b) = O(b^{d+1})$

Space complexity: If each node is retained in memory space complexity will be same as that of time complexity i.e. $O(b^{d+1})$

Optimality: Yes unless actions have different cost.

Advantages:

1. BFS will not get trapped anywhere .This contrasts with DFS which may follow a single unfruitful path for a very long time before the path actually terminates in a state that has no successors.
2. If there is solution then BFS is guaranteed to find it. Furthermore if there are multiple solutions then a minimal solution will be found. This is guaranteed by

the fact that longer paths are never explored until all shorter ones have already been examined. This contrasts with DFS which may find a long path to a solution in one part of the tree when a shorter path exists in some other unexplored part of the tree.

Disadvantages:

1. High storage requirement: *exponential* with tree depth.

2. Uniform Cost Search

Uniform-cost search is similar to breadth-first search. BFS is optimal when all the step costs are equal, 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 UCS expands the node with lowest path cost. UCS is the same as BFS when all step-costs are equal. This is implemented by means of a queue ordered by path cost.

We associate with each node *n* a cost *g(n)* that measures the cost of getting to node *n* from the *start* node. *g(start) = 0*. If n_i is a successor of *n*, then $g(n_i) = g(n) + c(n,n_i)$, where $c(n,n_i)$ is the cost of going from node *n* to node n_i .

Completeness: YES, if step-cost > ε (small positive constant)

Time complexity: Assume C* the cost of the optimal solution. Assume that every action costs at least ε. In the Worst-case time complexity will be O ($b^{C^*/\varepsilon}$)

Space complexity: Same as that of time complexity

Optimality: It will be optimal since nodes expanded in order of increasing path cost. (Also if complete)

Advantage:

1. Guaranteed to find the least-cost solution.

Disadvantages:

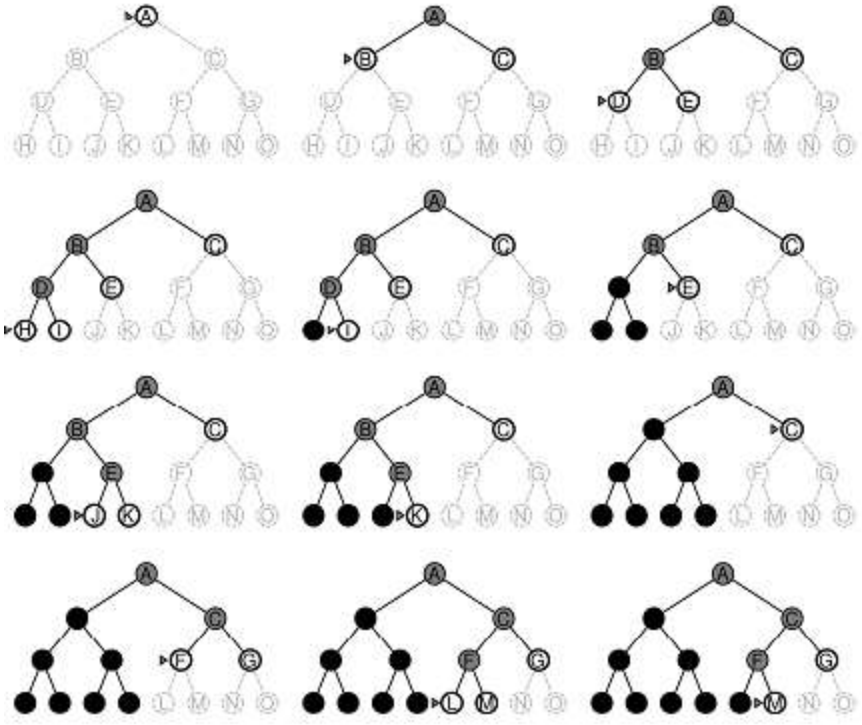
1. Exponential storage required.
2. `open` list must be kept sorted (as a priority queue).
3. Must change cost of a node on `open` if a lower-cost path to it is found.

Uniform-cost search is the same as Heuristic Search when no heuristic information is available (heuristic function *h* is always *0)*.

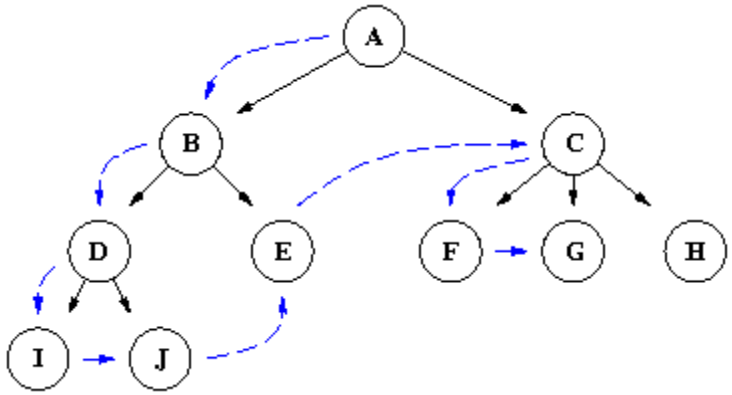
3. Depth First Search

DFS is another control strategy which always expands the deepest node. We could pursue a single branch of the tree until it yields a solution or until a decision to terminate the path is made. In depth-first search, the search is conducted from the start state as far as it can be carried until either a dead-end or the goal is reached. If a dead-end is reached, backtracking occurs.

The most recently created state from which alternative moves are available will be revisited and a new state is 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 steps where originally made. Specifically the most recent step is always the first to be undone. This form of backtracking is what is usually meant by the simple backtracking. An example is shown below:



In a depth-first search, after operators are applied to a state, one of the resultant states is considered next. So the order in which the states of the simple example space will be considered is:



Algorithm:

1. If the initial state is a goal state, terminate with success and return that state.

2. Otherwise do the following until success or failure is signaled,

   a) Generate a successor E of the initial state. If there are no more successors signal failure.
   b) Call DFS with E as initial state.
   c) If success is returned signal success otherwise continue in this loop

Usually DFS is implemented using a stack.(LIFO). In a stack, new elements are added to the front of the list rather than the back, but when the remove the paths to expand them, you still remove them from the front. The result of this is that DFS explores one path, ignoring alternatives, until it either finds the goal or it can't go anywhere else.

Completeness: Does it always find a solution if one exists? NO, unless search space is finite and no loops are possible.

Time complexity: $O(b^m)$ .Terrible if m is much larger than d (depth of optimal solution). But if many solutions are present, then faster than BF-search

Space complexity: O(bm) .Backtracking search uses even less memory. In that it generates only one successor instead of all

Optimality: No, unless search space is finite and no loops are possible.

Advantages:

1. DFS requires less memory since only the nodes on the current path are stored. This contrasts with BFS where all of the tree that has so far been generated must

be stored. Once a node has been expanded it can be removed from memory as soon as all its descendants have been fully explored.
2. Easily programmed: function call stack does most of the work of maintaining state of the search.
3. DFS may find a solution without examining much of the search space at all. This contrasts with BFS in which all parts of the tree must be examined to level 'n' before any nodes at level 'n+1' can be examined. This is particularly significant if many acceptable solutions exist. DFS can stop when one of them is found.

Disadvantages:

1. May find a sub-optimal solution (one that is deeper or more costly than the best solution).
2. Incomplete: without a depth bound, may not find a solution even if one exists.

A variant of DFS called **backtracking search** uses still lesser memory. In backtracking only one successor is generated at a time rather than all successors. Each partially expanded node remembers which successor to generate next. In this way only O(m) memory is needed rather than O(bm) where b is the branching factor and m is the maximum depth.

4. Depth Limited Search

Depth-first search can spend much time (perhaps infinite time) exploring a very deep path that does not contain a solution, when a shallow solution exists. An easy way to solve this problem is to put a maximum **depth limit** '*l*' on the search ie nodes at depth limit are treated as if they have no successors. Beyond the depth limit, a failure is generated automatically without exploring any deeper. This approach is called depth limited search. The depth limit solves the infinite path problem.

Unfortunately it also introduces an additional source of incompleteness if we choose $l < d$ ie shallowest goal is beyond the depth limit(if d is known).Depth limited search will not be optimal if we choose $l > d$. DFS can be viewed as a special case of Depth limited search with $l=\infty$.

If a solution exists at depth d or less, then d -limited DFS will find the solution, and the algorithm is very efficient in space. However, depth-limited DFS is not complete: If a solution exists but only at depth greater than d, then depth-limited DFS will not find the solution.

For eg : in the case of a TSP consider a map with 20 cities. Therefore we know that if there is a solution it must be of length 19 at the longest so $l = 19$ is a possible choice. But if we study the map carefully we can see that any city can be reached from any other city in at most 9 steps .This is known as diameter of state space gives us a better depth limit which leads to a more efficient depth limited search.

Depth limited search can terminate with 2 kinds of failure

1. standard failure indicates no solution
2. no solution within the depth limit

Algorithm :

1. Determine the vertex where the search should start and assign the maximum search depth
2. Check if the current vertex is within the maximum search depth
    - If not: Do nothing
    - If yes:
        1. Expand the vertex and save all of its successors in a stack
        2. Call DLS recursively for all vertices of the stack and go back to Step 2

Completeness : In general the algorithm is not complete since it does not find any solution that lies beyond the given search depth. But if you choose the maximum search depth to be greater than the depth of a solution the algorithm becomes complete.

Time complexity: $O(b^l)$ where l is the depth limit

Space complexity: O(bl) where l is the depth limit

Optimality: Depth-limited search is not optimal. It still has the problem of depth-first search that it first explores one path to its end, thereby possibly finding a solution that is more expensive than some solution in another path.

Problems:

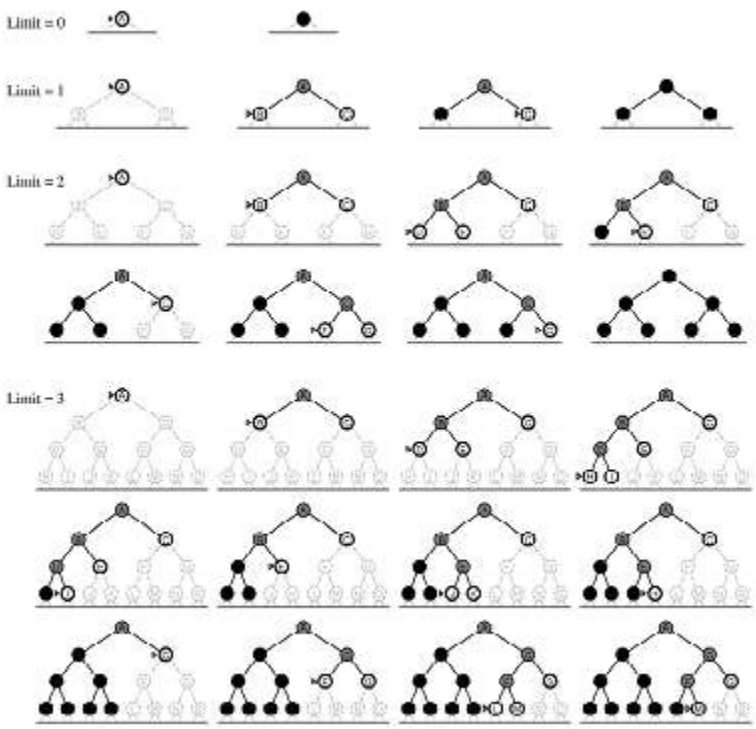
1. It's hard to guess how deep the solution lies.
2. If the estimated depth is too deep (even by 1) the computer time used is dramatically increased, by a factor of b^{extra}.
3. If the estimated depth is too shallow, the search fails to find a solution; all that computer time is wasted.

5. Depth First Iterative Deepening

Iterative deepening depth-first search or **IDDFS** is a state space search strategy in which a depth-limited search is run repeatedly, increasing the depth limit with each iteration until it reaches *d*, the depth of the shallowest goal state. On each iteration, the IDDFS visits the nodes in the search tree in the same order as depth-first search, but the cumulative order in which nodes are first visited, is effectively breadth-first.

Iterative deepening depth-first search combines depth-first search's space-efficiency and both breadth-first search's completeness (when the branching factor is finite) and optimality (when the path cost is a non-decreasing function of the depth of the node).Since iterative deepening visits states multiple times it may seem wasteful, but it turns out to be not so costly, since in a tree most of the nodes are in the bottom level, so it does not matter much if the upper levels are visited multiple times.

The main advantage of this algorithm in game tree searching is that the earlier searches tend to improve the commonly used heuristics, such as the killer heuristic and alpha-beta pruning, so that a more accurate estimate of the score of various nodes at the final depth search can occur, and the search completes more quickly since it is done in a better order (it can be shown that alpha-beta pruning works much better if the moves are generated with the moves likely to be the best moves evaluated first). An eg : is shown below:



In an iterative deepening search, the nodes on the bottom level are expanded once, those on the next to bottom level are expanded twice, and so on, up to the root of the search tree, which is expanded d + 1 times. So the total number of expansions in an iterative deepening search is

$N(IDS)=(d + 1)1 + (d)b + (d- 1)b^2 + \bullet\bullet\bullet + 3b^{d-2} + 2b^{d-1} + b^d$

Node generation:

level d: once

level d-1: 2
level d-2: 3
…….
level 2: d-1
level 1: d

$N(BFS)= b+b^2 +b^3 +...+ b^d + (b^{d+1} -b)$

In general, iterative deepening is the preferred search method when there is a large search space and the depth of the solution is not known. Iterative deepening essentially throws away the results of every depth-limited search before going on to the next. This means that iterative deepening simulates breadth-first search, but with only linear space complexity.

Similar to iterative deepening is a search strategy called **iterative lengthening search** that works with increasing path-cost limits instead of depth-limits. It turns out however, that iterative lengthening incurs substantial overhead that make it less useful than iterative deepening as a search strategy.

Algorithm :

1. Set DEPTH-LIMIT = 0
2. Conduct depth-first search to a depth of DEPTH-LIMIT If a solution path is found, then return it.
3. Increment DEPTH-LIMIT by 1 and go to step 2.

Completeness: YES (no infinite paths)

Time Complexity: Asymptotically approaches $O(b^d)$. Algorithm seems costly due to repeated generation of certain states.

Space Complexity: O(bd)

Optimality: YES if step cost is 1.

Advantages:

1. Finds an optimal solution (shortest number of steps).
2. Has the low (linear in depth) storage requirement of depth-first search.

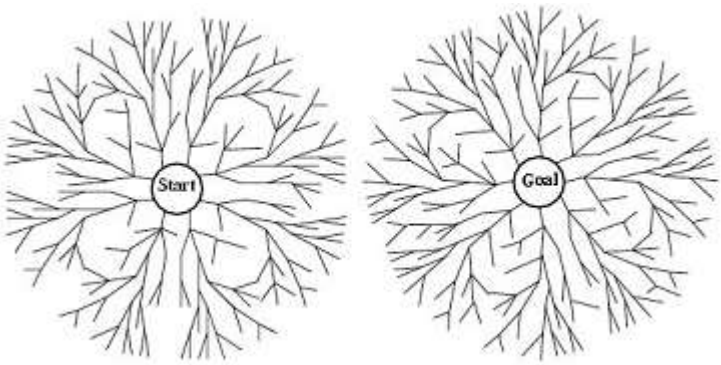
Disadvantage:

1. Some computer time is wasted re-exploring the higher parts of the search tree. However, this actually is not a very high cost.

6. Bi-Directional Search

Bidirectional search is a graph search algorithm that runs two simultaneous searches: one forward from the initial state and one backward from the goal and stopping when the two meet in the middle. It requires explicit goal state rather than a goal criterion. The reason for this approach is that each of the two searches has complexity $O(b^{d/2})$ (in Big O notation), and $O(b^{d/2} + b^{d/2})$ is much less than the running time of one search from the beginning to the goal, which would be $O(b^d)$.

ie, $b^{d/2} + b^{d/2} \neq b^d$



This does not come without a price: Aside from the complexity of searching two times in parallel, we have to decide which search tree to extend at each step; we have to be able to travel backwards from goal to initial state - which may not be possible without extra work; and we need an efficient way to find the intersection of the two search trees. This additional complexity means that the A* search algorithm is often a better choice if we have a reasonable heuristic.

It is implemented by having one or both of the searches check each node before it is expanded to see if it is in the fringe of the other search tree. At least one of the search tree needs to be kept in memory so that membership check can be done hence the space complexity is also $O(b^{d/2})$.This space requirement is the most significant weakness of bidirectional search. The algorithm is complete and optimal if both searches are BF.

Completeness: Yes

Time Complexity: $O(b^{d/2})$ like breadth first but to half depth

Space Complexity: $O(b^{d/2})$ need to retain at least that many nodes to guarantee completeness

Optimal: Yes if step cost is 1

Comparison of different search algorithms

Criterion	Breadth-First	Uniform-cost	Depth-First	Depth-limited	Iterative deepening	Bidirectional search
Complete?	YES*	YES*	NO	YES, if $l \geq d$	YES	YES*
Time	b^{d+1}	$b^{C^*/\varepsilon}$	b^m	b^l	b^d	$b^{d/2}$
Space	b^{d+1}	$b^{C^*/\varepsilon}$	bm	bl	bd	$b^{d/2}$
Optimal?	YES*	YES*	NO	NO	YES	YES

b = branching factor
d = depth of the shallowest solution
m = maximum depth of the search tree
l = depth limit
ε = step-cost
C* = cost of the optimal solution

Hill Climbing

Hill climbing is a **variant of generate and test** in which feedback from the test procedure is used to help the generator decide which direction to move in the search space. In a pure generate and test procedure the test function responds with only a "yes" or "no". But if the test function is augmented with a **heuristic function** that provides an estimate of a how close a given state is to a goal state. This is particularly nice because often the computation of the heuristic function can be done at almost no cost at the same time that the test for a solution is being performed. Hill climbing is often used when a good heuristic function is available for evaluating states but when no other useful knowledge is available.

Hill climbing is an **optimization** technique which belongs to the family of Local search (optimization).Hill climbing attempts to **maximize** (or minimize) a **function** $f(x)$, where x are discrete states. These states are typically represented by vertices in a graph, where edges in the graph encode nearness or similarity of a graph. Hill climbing will follow the graph from vertex to vertex, always locally increasing (or decreasing) the value of f, until a **local maximum** x_m is reached.

1) Simple Hill Climbing

In hill climbing the basic idea is to always head towards a state which is better than the current one. So, if you are at town A and you can get to town B and town C (and your target is town D) then you should make a move IF town B or C appear nearer to town D than town A does.

Algorithm:

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise continue with initial state as the current state
2. Loop until a solution is found or there are no new operators left to be applied in the current state:
    a) Select an operator that has not yet been applied to the current state and apply it to produce a new state
    b) Evaluate the new state:
        i. If it is a goal state, then return it and quit.
        ii. If it is not a goal state but it is better than current state, then make it the current state
        iii. If it is not better than the current state, then continue in the loop

The key difference between this algorithm and the one we gave for generate and test is the use of an evaluation function as a way to inject task specific knowledge into the control process. For the algorithm to work, a precise definition of better must be provided. In some cases, it means a higher value of heuristic function. In others it means a lower value.

Note that the algorithm does not attempt to exhaustively try every node and path, so no node list or agenda is maintained - just the current state. If there are loops in the search space then using hill climbing you shouldn't encounter them - you can't keep going up and still get back to where you were before. Remember, from any given node, we go to the first node that is better than the current node. If there is no node better than the present node, then hill climbing halts.

2) Steepest-Ascent Hill Climbing

A useful variation on simple hill climbing considers all the moves from the current state and **selects the best one as the next state**. This method is called **Steepest-Ascent Hill Climbing or Gradient Search.** This is a case in which hill climbing can also operate on a continuous space: in that case, the algorithm is called **gradient ascent** (or gradient descent if the function is minimized).Steepest ascent hill climbing is similar to **best first search** but the latter tries all possible extensions of the current path in order whereas simple hill climbing only tries one. Following the steepest path might lead you to the goal faster, but then again, hill climbing does not guarantee you will find a goal.

Algorithm:

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise continue with initial state as the current state
2. Loop until a solution is found or until a complete iteration produces no change to the current state:

    a) Let SUCC be a state such that any possible successor of the current state will be better than SUCC
    b) For each operator that applies to the current state do:

        i. Apply the operator and generate the new state.
        ii. Evaluate the new state. If it is a goal state, then return it and quit. If it is not a goal state, compare it to SUCC. If it is better, then set SUCC to this state. If it is not better, leave SUCC alone.

    c) If the SUCC is better than the current state, then set the current state to SUCC

In **simple hill climbing**, the first closer node is chosen whereas in **steepest ascent hill climbing** all successors are compared and the closest to the solution is chosen. Both forms fail if there is no closer node. This may happen if there are local maxima in the search space which are not solutions. **Both** basic and steepest ascent hill climbing may **fail to find a solution**. Either algorithm may terminate not by finding a goal state but by getting to a state from which no better states can be generated. This will happen if the program has reached either a **local maxima, a ridge or a plateau**.

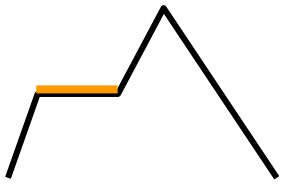
Problems

1) Local Maxima

A problem with hill climbing is that it will find only local maxima. **Local maximum** is a state that is better than all of its neighbours, but is not better than some other states far away. At a local maximum, all moves appear to make things worse. Local maxima are particularly frustrating because they often occur within sight of a solution. In this case they are called **foothills**. Unless the heuristic is good / smooth, it need not reach a global maximum. Other local search algorithms try to overcome this problem such as stochastic hill climbing, random walks and simulated annealing. One way to solve this problem with standard hill climbing is to iterate the hill climbing.

2) Plateau

Another problem with Hill climbing is called the **Plateau** problem. This occurs when we get to a "flat" part of the search space, i.e. we have a path where the heuristics are all very close together and we end up wandering aimlessly. It is a **flat area** of search space in which a whole set of neighboring states have the **same value**. On a plateau it is not possible to determine the best direction in which to move by making local comparisons.



3) Ridges

A ridge is a curve in the search place that leads to a maximum. But the orientation of the ridge compared to the available moves that are used to climb is such that each move will lead to a smaller point. In other words to the algorithm each point on a ridge looks a like a local maximum even though the point is part of a curve leading to a better optimum. It is an area of the search space that is higher than surrounding areas and that itself has a slope. However, two moves executed serially may increase the height.

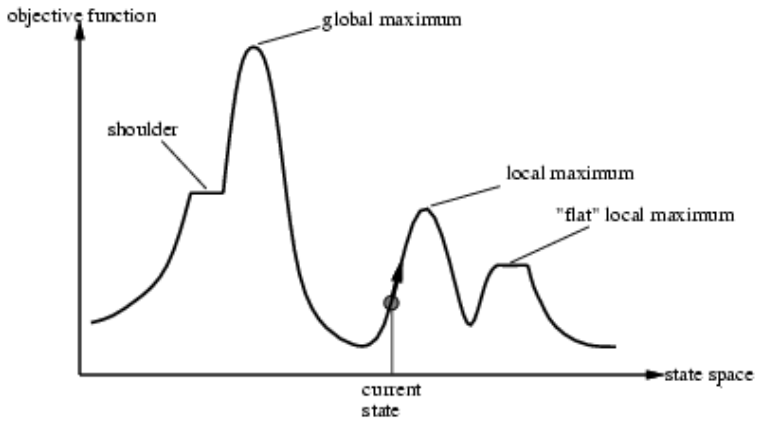


Figure above shows a one dimensional state space landscape in which elevation corresponds to the objective function or heuristic function. The aim is to find the global maxima. Hill climbing search modifies the current state to try to improve it, as shown by the arrow.

Some ways of dealing with these problems are:

- **Backtrack** to some earlier node and try going in a different direction. This is particularly reasonable if at that node there was another direction that was looked as promising or almost as promising as the one that was chosen earlier. To implement this strategy, maintain a list of paths almost taken and go back to one of them if the path that was taken leads to a dead end. This is a fairly good way of dealing with local maxima.

- **Make a big jump in some direction** to try to get to a new section of the search space. This is particularly good way of dealing with plateaus. If the only rules available describe single small steps, apply them several times in the same direction.

- **Apply 2 or more rules before doing the test**. This corresponds to moving in several directions at once. This is particularly good strategy of dealing with ridges.

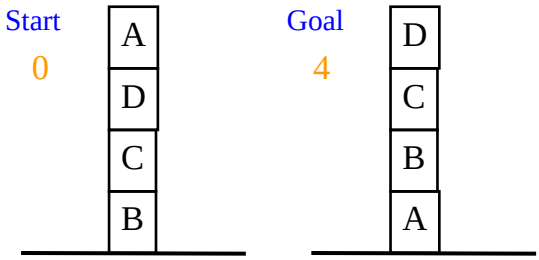
Even with these first aid measures, hill climbing is not always very effective. Hill climbing is a local method and it decides what to do next by looking only at the "immediate" consequences of its choices. Although it is true that hill climbing procedure itself looks only one move ahead and not any farther, that examination may in fact exploit an arbitrary amount of global information if that information is encoded in the heuristic function. Consider the blocks world problem shown in figure below. Assume the operators:

1. Pick up one block and put it on the table
2. Pick up one block and put it on another one

Suppose we use the following heuristic function:

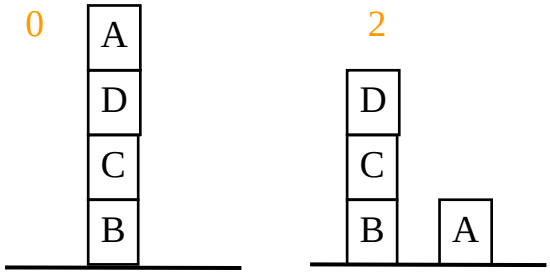
Local heuristic:

1. Add 1 point for every block that is resting on the thing it is supposed to be resting on.
2. Subtract 1 point for every block that is resting on a wrong thing.

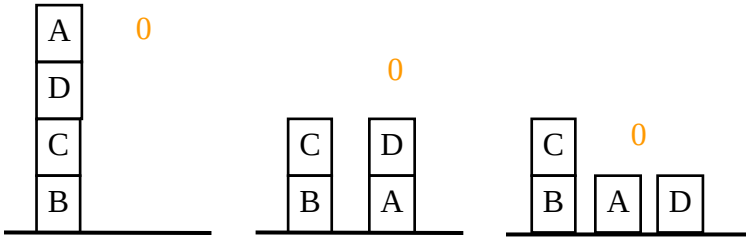


Blocks World

Using this function, the goal state has a score of 4. The initial state has a score of 0 (since it gets 1 point added for blocks C and D and one point subtracted for blocks A and B). There is only 1 move from initial state, namely to move block A to the table. That produces a state with a score of 2 (since now A's position causes a point to be added than subtracted). The hill climbing will accept that move.



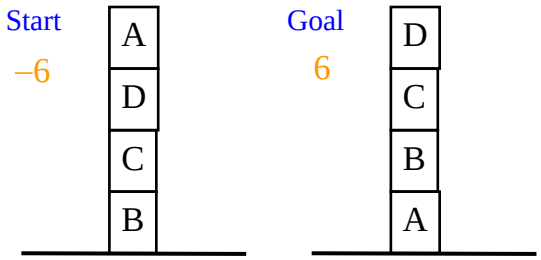
From the new state there are 3 possible moves, leading to the 3 states shown in figure below. These states have scores: (a) 0, (b) 0, and (c) 0. Hill climbing will halt because all these states have lower scores than the current state. This process has reached a local maximum that is not the global maximum.



The problem is that by purely local examination of local structures, the current state appears to be better than any of its successors because more blocks rest on the correct objects. Suppose we try the following heuristic function in place of first one:

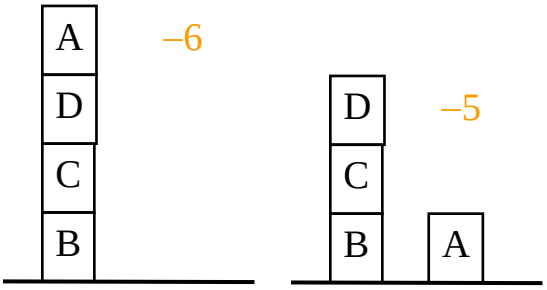
Global heuristic:

1. For each block that has the correct support structure (i.e. the complete structure underneath it is exactly as it should be), add 1 point to every block in the support structure.
2. For each block that has a wrong support structure, subtract 1 point for every block in the existing support structure.

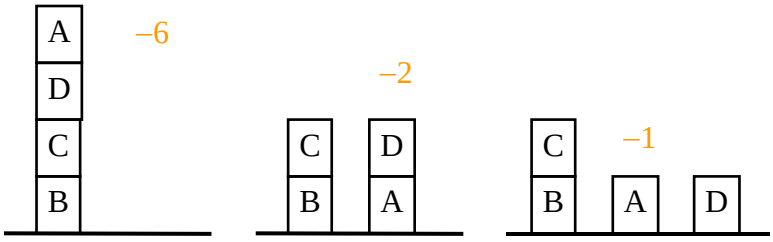


Blocks World

Using this heuristic function, the goal state has the score 6(1 for B, 2 for C, etc.). The initial state has the score -6. Moving A to the able yields a state with a score of -5 since A no longer has 3 wrong blocks under it.



The 3 new states that can be generated next now have the following scores: (a) -6,(b) -2, and (c) -1. This time steepest ascent hill climbing will chose move (c), which is the correct one.



This new heuristic function captures the 2 key aspects of this problem: incorrect structures are bad and should be taken apart; and correct structures are good and should be built up. As a result the same hill climbing procedure that failed with the earlier heuristic function now works perfectly. Unfortunately it is not always possible to construct such a perfect heuristic function. Often useful when combined with other methods, getting it started right in the right general neighbourhood.

Hill climbing is used widely in artificial intelligence fields, for reaching a goal state from a starting node. Choice of next node/ starting node can be varied to give a list of related algorithms. Some of them are:

1) **Random-restart hill climbing**

Random-restart hill climbing is a meta-algorithm built on top of the hill climbing algorithm. It is also known as **Shotgun hill climbing**. Random-restart hill climbing simply runs an outer loop over hill-climbing. Each step of the outer loop chooses a random initial condition x_0 to start hill climbing. The best x_m is kept: if a new run of hill climbing produces a better x_m than the stored state, it replaces the stored state.

It conducts a series of hill climbing searches from randomly generated initial states stopping when a goal is found. Random-restart hill climbing is a surprisingly effective algorithm in many cases. It turns out that it is often better to spend CPU time exploring the space, rather than carefully optimizing from an initial condition.

2) **Stochastic hill climbing**

Stochastic hill climbing chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move. This usually converges more slowly than the steepest ascent, but in some state landscapes it finds better solutions.

3) **First-choice hill climbing**

First-choice hill climbing implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state. This is a good strategy when a state has many (e.g., thousands) of successors.

Simulated Annealing

Simulated Annealing is a **variation of hill climbing** in which, at the beginning of the process, some **downhill** moves may be made. The idea is to do enough exploration of the whole space early on, so that the final solution is relatively insensitive to the starting state. This should lower the chances of getting caught at a local maximum, or plateau, or a ridge.

In order to be compatible with standard usage inn discussions of simulated annealing we make 2 notational changes for the duration of this section. We use the term **objective function** in place of the term heuristic function. And we attempt to **minimize** rather than maximize the value of the objective function.

Simulated Annealing as a computational process is patterned after the physical process of **annealing** in which **physical substances** such as metals are **melted** (i.e. raised to high energy levels) and then **gradually cooled** until some solid state is reached. The goal of the process is to produce a **minimal-energy final** state. Physical substances usually move from higher energy configurations to lower ones, so the valley descending occurs naturally. But there is some **probability** that a **transition** to a **higher energy state** will **occur**. This probability is given by the function

$$\mathbf{p = e^{-\Delta E/kT}}$$

where ΔE = positive change in energy level
T = temperature
k = Boltzmann's constant

Thus in the physical valley descending that occurs during annealing, the probability of a large uphill move is lower than the probability of a small one. Also, the probability that an uphill move will be made decreases as the temperature decreases. Thus such moves are more likely during the beginning of the processes when the temperature is high, and they become less likely at the end as the temperature becomes lower.

The rate at which the system is cooled is called **annealing schedule**. Physical annealing processes are very sensitive to the annealing schedule. If cooling occurs too rapidly, stable regions of high energy will form. In other words a local but not global minimum is reached. If however a slower schedule is used a uniform crystalline structure which corresponds to a global minimum is more likely to develop.

These properties of physical annealing can be used to define an analogous process of simulated annealing which can be used whenever simple hill climbing can be used. In this analogous process ΔE is generalized so that it represents not specifically the change in energy but more generally the change in the value of the **objective function**. The variable k describes the correspondence between the units of temperature and the units of energy. Since in this analogous process the units of both T and E are artificial it makes sense to incorporate k into T, selecting values for T that produce desirable behaviour on the part of the algorithm. Thus we used the revised probability formula

$$\mathbf{p^1 = e^{-\Delta E/T}}$$

The algorithm for simulated annealing is only slightly different from the simple hill climbing procedure. The three differences are:

- The annealing schedule must be maintained
- Move to worse states may be accepted

- It is a god idea to maintain, in addition to the current state, the best state found so far. Then if the final state is worse than the earlier state, the earlier state is till available.

Algorithm:

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise continue with initial state as the current state.
2. Initialize BEST-SO-FAR to the current state.
3. Initialize T according to the annealing schedule.
4. Loop until a solution is found or there are no new operators left to be applied in the current state:
    a) Select an operator that has not yet been applied to the current state and apply it to produce a new state.
    b) Evaluate the new state. Compute
    ΔE = Value of current state – Value of new state
        i. If the new state is a goal state, then return it and quit.
        ii. If it is not a goal state but it is better than the current state, then make it the current state. Also set BEST-SO-FAR to this new state.
        iii. If it is not better than the current state, then make it the current state with the probability p^1 as defined above. This step is usually implemented by invoking a random number generator to produce a number in the range [0,1]. If that number is less than p^1, then the move is accepted. Otherwise, do nothing.
    c) Revise T as necessary according to the annealing schedule
5. Return BEST-SO-FAR as the answer.

To implement this revised algorithm, it is necessary to select an annealing schedule which has 3 components.

1) The initial value to be used for temperature
2) The criteria that will be used to decide when the temperature of the system should be reduced
3) The amount by which the temperature will be reduced each time it is changed.

There may also be a fourth component of the schedule namely when to **quit**. Simulated annealing is often used to solve problems in which number of moves from a given state is very large. For such problems, it may not make sense to try all possible moves. Instead, it may be useful to exploit some criterion involving the number of moves that have been tried since an improvement was found.

While designing a schedule the first thing to notice is that as T approaches zero, the probability of accepting a move to a worse state goes to zero and simulated annealing becomes identical to hill climbing. The second thing to notice is that what really matters in computing the probability of accepting a move is the ratio $\Delta E/T$. Thus it is important that values of T be scaled so that this ratio is meaningful

3.1 GENERATE-AND-TEST

The generate-and-test strategy is the simplest of all the approaches we discuss. It consists of the following steps:

Algorithm: Generate-and-Test

1. Generate a possible solution. For some problems, this means generating a particular point in the problem space. For others, it means generating a path from a start state.
2. Test to see if this is actually a solution by comparing the chosen point or the endpoint of the chosen path to the set of acceptable goal states.
3. If a solution has been found, quit. Otherwise, return to step 1.

If the generation of possible solutions is done systematically, then this procedure will find a solution eventually, if one exists. Unfortunately, if the problem space is very large, "eventually" may be a very long time.

The generate-and-test algorithm is a depth-first search procedure since complete solutions must be generated before they can be tested. In its most systematic form, it is simply an exhaustive search of the problem space. Generate-and-test can, of course, also operate by generating solutions randomly, but then there is no guarantee that a solution will ever be found. In this form, it is also known as the British Museum algorithm, a reference to a method for finding an object in the British Museum by wandering randomly.[1] Between these two extremes lies a practical middle ground in which the search process proceeds systematically, but some paths are not considered because they seem unlikely to lead to a solution. This evaluation is performed by a heuristic function, as described in Section 2.2.2.

The most straightforward way to implement systematic generate-and-test is as a depth-first search tree with backtracking. If some intermediate states are likely to appear often in the tree, however, it may be better to modify that procedure, as described above, to traverse a graph rather than a tree.

For simple problems, exhaustive generate-and-test is often a reasonable technique. For example, consider the puzzle that consists of four six-sided cubes, with each side of each cube painted one of four colors. A solution to the puzzle consists of an arrangement of the cubes in a row such that on all four sides of the row one block face of each color is showing. This problem can be solved by a person (who is a much slower processor for this sort of thing than even a very cheap computer) in several minutes by systematically and exhaustively trying all possibilities. It can be solved even more quickly using a heuristic generate-and-test procedure. A quick glance at the four blocks reveals that there are more, say, red faces than there are of other colors. Thus when placing a block with several red faces, it would be a good idea to use as few of them as possible as outside faces. As many of them as possible should be placed to abut the next block. Using this heuristic, many configurations need never be explored and a solution can be found quite quickly.

Unfortunately, for problems much harder than this, even heuristic generate-and-test, all by itself, is not a very effective technique. But when combined with other techniques to restrict the space in which to search even further, the technique can be very effective.

For example, one early example of a successful AI program is DENDRAL [Lindsay *et al.*, 1980], which infers the structure of organic compounds using mass spectrogram and nuclear magnetic resonance (NMR) data. It uses a strategy called *plan-generate-test* in which a planning process that uses constraint-satisfaction techniques (see Section 3.5) creates lists of recommended and contraindicated substructures. The generate-and-test procedure then uses those lists so that it can explore only a fairly limited set of structures. Constrained in this way, the generate-and-test procedure has proved highly effective.

This combination of planning, using one problem-solving method (in this case, constraint satisfaction) with the use of the plan by another problem-solving method, generate-and-test, is an excellent example of the way techniques can be combined to overcome the limitations that each possesses individually. A major weakness of planning is that it often produces somewhat inaccurate solutions since there is no feedback from the world. But by using it only to produce pieces of solutions that will then be exploited in the generate-and-test process, the lack of detailed accuracy becomes unimportant. And, at the same time, the combinatorial problems that arise in simple generate-and-test are avoided by judicious reference to the plans.

Python

Python is a general-purpose interpreted, interactive, object-oriented and high-level programming language. Python was created by Guido van Rossum in the late eighties and early nineties. Like Perl, Python source code is also now available under the GNU General Public License (GPL).

This tutorial has been designed for software programmers with a need to understand the Python programming language starting from scratch. This tutorial will give you enough understanding on Python programming language from where you can take yourself to a higher level of expertise

Python Features:

Python's feature highlights include:

 Easy-to-learn: Python has relatively few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language in a relatively short period of time.

 Easy-to-read: Python code is much more clearly defined and visible to the eyes.

 Easy-to-maintain: Python's success is that its source code is fairly easy-to-maintain.

 A broad standard library: One of Python's greatest strengths is the bulk of the library is very portable and cross-platform compatible on UNIX, Windows and Macintosh.

 Interactive Mode: Support for an interactive mode in which you can enter results from a terminal right to the language, allowing interactive testing and debugging of snippets of code.

 Portable: Python can run on a wide variety of hardware platforms and has the same interface on all platforms.

 Extendable: You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.

 Databases: Python provides interfaces to all major commercial databases.

 GUI Programming: Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh and the X Window system of Unix.

 Scalable: Python provides a better structure and support for large programs than shell scripting

Install Python:

Python distribution is available for a wide variety of platforms. You need to download only the binary code applicable for your platform and install Python.

If the binary code for your platform is not available, you need a C compiler to compile the source code manually.

Compiling the source code offers more flexibility in terms of choice of features that you require in your installation. Here is a quick overview of installing Python on various platforms:

.

Setting path at Unix/Linux:

To add the Python directory to the path for a particular session in Unix:

 In the csh shell: type

setenv PATH "$PATH:/usr/local/bin/python" and press Enter.

 In the bash shell (Linux): type export PATH="$PATH:/usr/local/bin/python" and press Enter.

 In the sh or ksh shell: type PATH="$PATH:/usr/local/bin/python" and press Enter.

Python Lists:

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type.

The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator. For example:

```
#!/usr/bin/python

list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']

print list          # Prints complete list
print list[0]       # Prints first element of the list
print list[1:3]     # Prints elements starting from 2nd till 3rd
print list[2:]      # Prints elements starting from 3rd element
print tinylist * 2  # Prints list two times
print list + tinylist # Prints concatenated lists
```

This will produce the following result:

```
['abcd', 786, 2.23, 'john', 70.200000000000003]
abcd
[786, 2.23]
[2.23, 'john', 70.200000000000003]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john']
```

Python Tuples:

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.

The main differences between lists and tuples are: Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be thought of as **read-only** lists. For example:

```
#!/usr/bin/python

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
tinytuple = (123, 'john')

print tuple           # Prints complete list
print tuple[0]        # Prints first element of the list
print tuple[1:3]      # Prints elements starting from 2nd till 3rd
print tuple[2:]       # Prints elements starting from 3rd element
print tinytuple * 2   # Prints list two times
print tuple + tinytuple # Prints concatenated lists
```

This will produce the following result:

```
('abcd', 786, 2.23, 'john', 70.200000000000003)
abcd
(786, 2.23)
(2.23, 'john', 70.200000000000003)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john')
```

Following is invalid with tuple, because we attempted to update a tuple, which is not allowed. Similar case is possible with lists:

```
#!/usr/bin/python

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
list = [ 'abcd', 786 , 2.23, 'john', 70.2  ]
tuple[2] = 1000    # Invalid syntax with tuple
list[2] = 1000     # Valid syntax with list
```

Python Dictionary:

Python's dictionaries are kind of hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]). For example:

```
#!/usr/bin/python

dict = {}
dict['one'] = "This is one"
dict[2]     = "This is two"

tinydict = {'name': 'john','code':6734, 'dept': 'sales'}

print dict['one']       # Prints value for 'one' key
print dict[2]           # Prints value for 2 key
print tinydict          # Prints complete dictionary
print tinydict.keys()   # Prints all the keys
print tinydict.values() # Prints all the values
```

This will produce the following result:

```
This is one
This is two
{'dept': 'sales', 'code': 6734, 'name': 'john'}
['dept', 'code', 'name']
['sales', 6734, 'john']
```

Dictionaries have no concept of order among elements. It is incorrect to say that the elements are "out of order"; they are simply unordered.

Python implementation of hill climbing

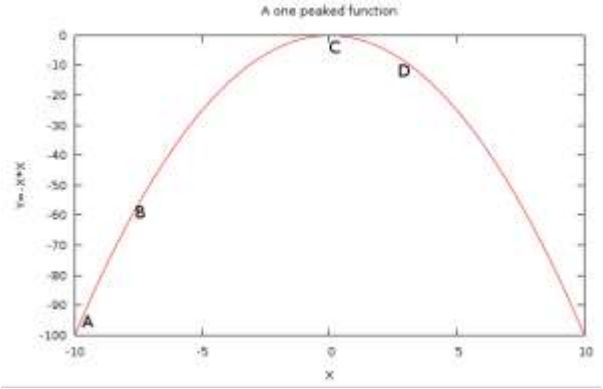
An optimization problem can usually also be modelled as a search problem, since we are searching for the optimum solution from among the solution space. Without any loss of generality, we can assume that our optimization problems are of the maximization category.

So, here is the hill climbing *technique* of search:

1. Start with an initial solution, also called the *starting point.* Set *current point* as the starting point
2. Make a move to a next solution, called the *move operation*
3. If the move is a good move, then set the new point as the *current point* and repeat (2). If the move is a bad move, terminate. The last current solution is the possible optimum solution.

Move operation: The move operation is problem dependent. In a discrete optimization problem, such as the Travelling Salesman Problem, a move

operation would probably shuffle a couple of positions in the original solution. In problems where the search space is continuous, techniques such as those used in steepest ascent/descent methods are used to obtain the next solution.
Good/Bad Move: A move is said to be good, if a point obtained by the move operation improves the quality of the solution, as compared to the previous solution. A bad move is defined similarly.
Quality of solution: In all optimization problems, there is always atleast on quantity (in single-objective optimization problems) that we want to improve. A quality of solution is judged on the basis of this quantity. In a function maximization problem, the function value itself is a measure of the quality.
Illustration:
Let us now consider a simple one-peaked function f(x) = -x*x and we seek the maximum of this function. This is a plot of the function:

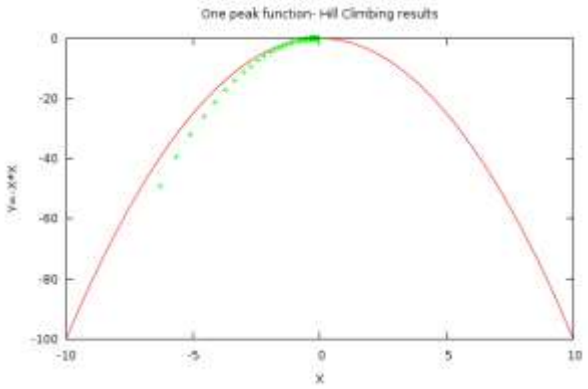


Sample one-peak function

- Let A be the starting point, and let the move operation take us to point B (pretty far away; usually it will be much closer to A but it will serve the purpose here). Consider the quality of the two solutions, A & B
- B definitely improves our search, since the function value at B is more than A, we keep moving, setting B as the current point
- Our next move operation takes us to C (our desired solution, but we don't know that yet). Now, compare B & C. Since C is definitely a good move, so, we set the current point to C and keep moving

- Our next move takes us to D. On comparing C &D, we see that it was a bad move, Hence we terminate. Hence, the current point, C is our solution.

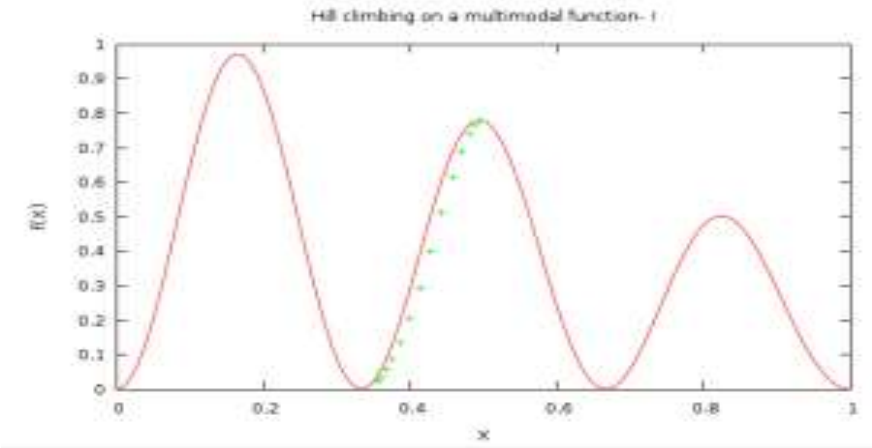
The following plot shows the results obtained via simple Hill-climbing on the above function:



Hill clmbing results on a simple one-peak function (The green points are the ones obtained via a Hillclimbing search)

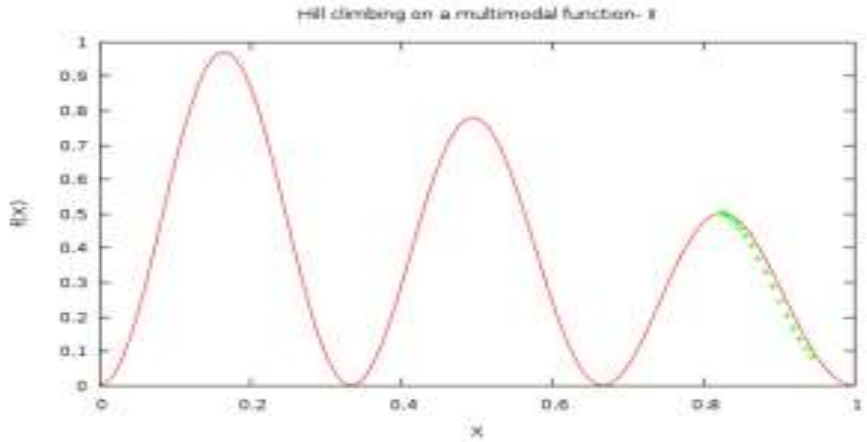
Simplicity is a hallmark of the Hill climbing search procedure. However, its simplicity comes at a price. In a multi-modal landscape, Hill climbing procedure will terminate the first peak it finds. It may be the global maximum, but it could also be a local maxima. This is illustrated below:

Case I:



Hill Climbing on a multimodal function – I

Case II:



Hill climbing on multimodal function- II

As you can see from the figures above, depending on your starting point, you will reach the nearest peak of your function. That is definitely a weak point in a multimodal landscape. To overcome this limitation random restart hill climbing algorithm, i.e with a random initial point has been proposed (by who?).

Hill climbing has been used on its own for many optimization problems as well as as a local search operator in global search algorithms, like Evolutionary Algorithms (GAs, ES, etc).

This is a simple Python implementation of hill climbing that I used for illustration. It contains some of the functions that I plotted above as well.

Hill-climbing with Multiple Solutions

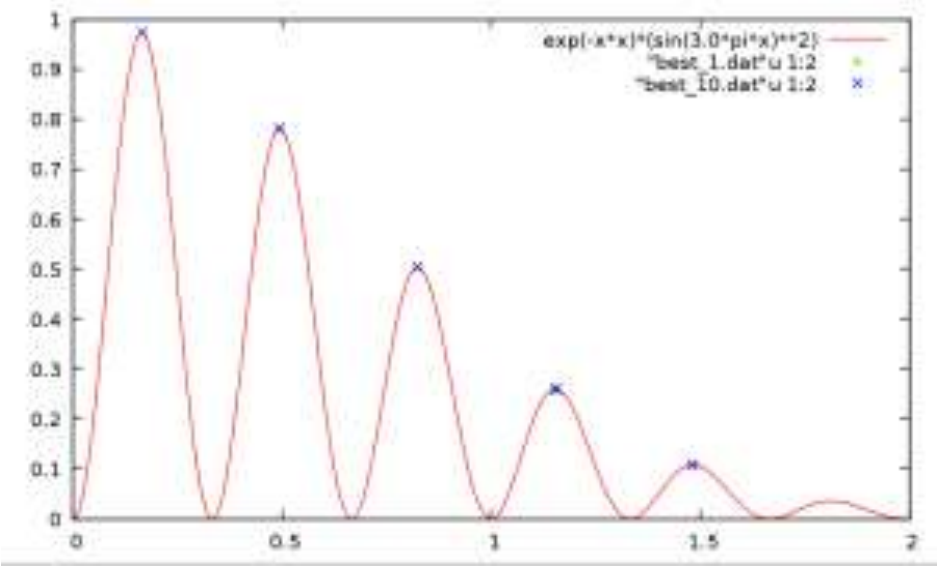
As you have noticed earlier, the classic hill climbing will not go beyond the first peak it reaches. In a multi-modal landscape this can indeed be limiting. I made some simple changes to the above algorithm to allow hill-climbing to go beyond the first peak it reaches. They are:

- **Multiple starting points**: Instead of searching for the highest peak, starting from a single peak, search from multiple points. This simulates a ***parallel search,***

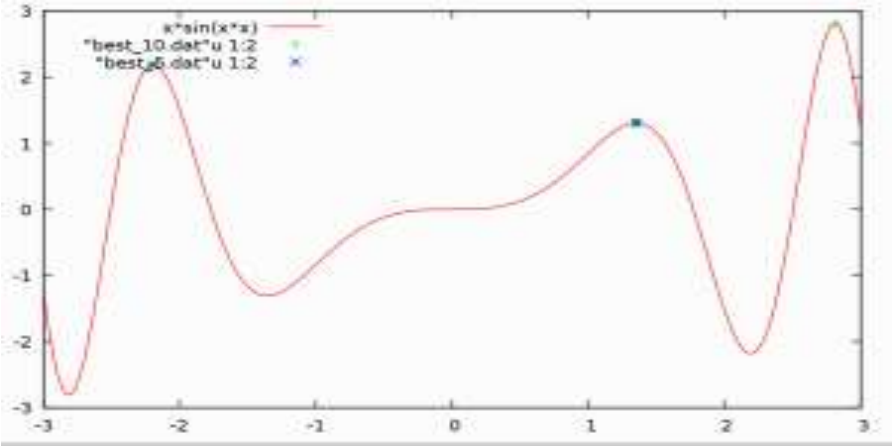
similar in principle to the way population based algorithms function. This is also sometimes referred to as **a *random-restart* hill-climbing.**

- **Random move when stuck**: If a particular point gets stuck on its uphill climb, a random noise is added to it. A point is allowed to be stuck for a specified number of times, before beginning with the next point, if any.

With these changes, I saw that the algorithm is now able to find the highest peak with two simple 1-D functions:



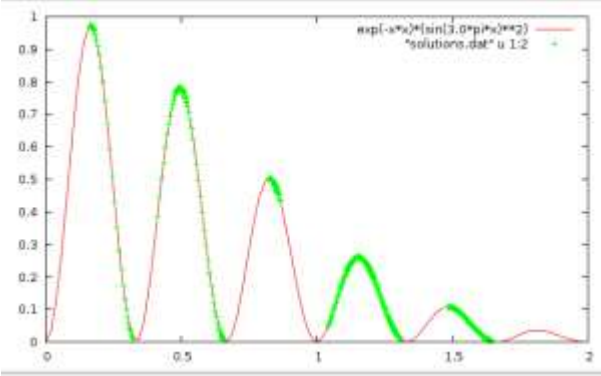
Sample Function-1



Sample Function – 2

The updated code is now available here Some other minor changes in the code are given as comments. The code also reports the best solution reached with each of the multiple starting points: (the output below is for the sample function 1)

One final plot to show how the multiple points searched the peaks:



Solutions from each point goes to its nearest peak

Solutions from each point goes to its nearest peak, effectively forming a parallel search. Its interesting to note that how it finds all the peaks. This is a very naive way of finding all the optima in a Multi-modal optimization algorithm.