UNIT-I

1. INTRODUCTION:

1.1 what is intelligence:

"The capacity to learn and solve problems"

In particular,

- The ability to solve novel problems
- The ability to act rationally
- The ability to act like humans

1.2 WHAT IS ARTIFICIAL INTELLIGENCE?

Artificial Intelligence 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.

- **Yes, but what is Intelligence?**

  Intelligence is the computational part of the ability to achieve goals in the world.

- **Isn't there a solid definition of intelligence that doesn't depend on relating it to human intelligence?**

Not yet. The problem is that we cannot yet characterize in general what kinds of computational procedures we want to call intelligent. We understand some of the mechanisms of intelligence and not others.

- **What's involved in Intelligence?**

- Ability to interact with the real world
  - to perceive, understand, and act
  - e.g., speech recognition and understanding and synthesis
  - e.g., image understanding
  - e.g., ability to take actions, have an effect
  - Reasoning and Planning
  - modeling the external world, given input
  - solving new problems, planning, and making decisions
  - ability to deal with unexpected problems, uncertainties
- Learning and Adaptation

- we are continuously learning and adapting
- our internal models are always being "updated"
  - e.g., a baby learning to categorize and recognize animals

1.3 BRIEF HISTORY OF AI

1.3.1 AI prehistory:

Philosophy:

Logic, methods of reasoning, mind as physical system foundations of learning, language, rationality.

Mathematics:

Formal representation and proof algorithms, computation, (un)decidability (in)tractability, probability

Probability/Statistics:

Modeling uncertainty, learning from data.

Economics:

Utility, decision theory, rational economic agents

Neuroscience :

Neurons as information processing units.

Psychology/Cognitive Science:

How do people behave, perceive, process cognitive information and represent knowledge.

Computer engineering: building fast computers

Control theory: design systems that maximize an objective function over time

Linguistics : knowledge representation, grammars

1.3.2 History of AI:

Intellectual roots of AI date back to the early studies of the nature of knowledge and reasoning. The dream of making a computer imitate humans also has a very early history.

The concept of intelligent machines is found in Greek mythology. There is a story in the 8th century A.D about Pygmalion Olio, the legendary king of Cyprus. He fell in love with an ivory statue he made to represent his ideal woman. The king prayed to the goddess Aphrodite, and the goddess miraculously brought the statue to life. Other myths involve human-like artifacts. As a present from Zeus to Europa, Hephaestus created Talos, a huge robot. Talos was made of bronze and his duty was to patrol the beaches of Crete.

Aristotle (384-322 BC) developed an informal system of syllogistic logic, which is the basis of the first formal deductive reasoning system.

Early in the 17th century, Descartes proposed that bodies of animals are nothing more than complex machines.

Pascal in 1642 made the first mechanical digital calculating machine.

In the 19th century, George Boole developed a binary algebra representing (some) "laws of thought."

Charles Babbage & Ada Byron worked on programmable mechanical calculating machines.

1943: early beginnings: McCulloch & Pitts: Boolean circuit model of brain.

1950: Turing: Turing's "Computing Machinery and Intelligence".

1956: birth of AI: Dartmouth meeting: "Artificial Intelligence" name adopted.

1950s: initial promise:

Early AI programs, including Samuel's checkers program Newell & Simon's Logic Theorist

1955-65: "great enthusiasm":

- Newell and Simon: GPS, general problem solver
- Gelertner: Geometry Theorem Prover
- McCarthy: invention of LISP

1966—73: Reality dawns

- Realization that many AI problems are intractable
- Limitations of existing neural network methods identified
  - Neural network research almost disappears

1969—85: Adding domain knowledge

- Development of knowledge-based systems
- Success of rule-based expert systems,
  - E.g., DENDRAL, MYCIN
  - But were brittle and did not scale well in practice

1986-- Rise of machine learning

- Neural networks return to popularity
- Major advances in machine learning algorithms and applications

1990-- Role of uncertainty

- Bayesian networks as a knowledge representation framework

1995-- AI as Science

- Integration of learning, reasoning, knowledge representation
- AI methods used in vision, language, data mining, etc

1.3.3 Can we build hardware as complex as the brain?

How complicated is our brain?

- a neuron, or nerve cell, is the basic information processing unit
- estimated to be on the order of 10^{12} neurons in a human brain
- many more synapses (10^{14}) connecting these neurons
- cycle time: 10^{-3} seconds (1 millisecond)

How complex can we make computers?

- 10^8 or more transistors per CPU
- supercomputer: hundreds of CPUs, 10^{12} bits of RAM
- cycle times: order of 10^{-9} seconds

Conclusion

YES: in the near future we can have computers with as many basic processing elements as our brain, but with

- Far fewer interconnections (wires or synapses) than the brain.
- Much faster updates than the brain.

But building hardware is very different from making a computer behave like a brain!

1.3.4 Can Computers Talk?

This is known as "speech synthesis"

- translate text to phonetic form

e.g., "fictitious" -> fik-tish-es

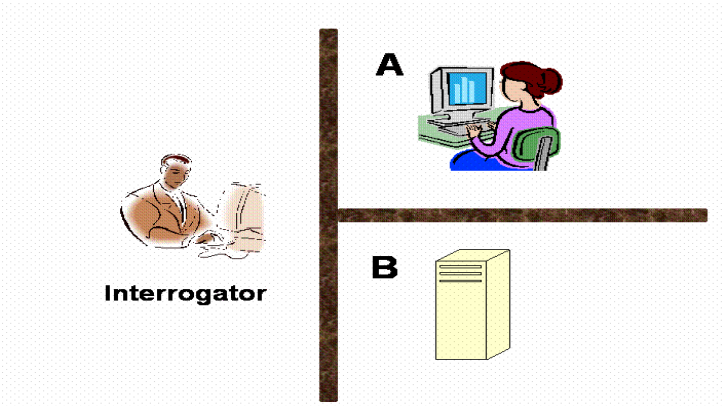
- use pronunciation rules to map phonemes to actual sound

e.g., "tish" -> sequence of basic audio sounds

- **Difficulties**
  - sounds made by this "lookup" approach sound unnatural
  - sounds are not independent
    - e.g., "act" and "action"
    - modern systems (e.g., at AT&T) can handle this pretty well
  - a harder problem is emphasis, emotion, etc
    - humans understand what they are saying
    - machines don't: so they sound unnatural
- **Conclusion:**
  - NO, for complete sentences.
  - YES, for individual words.

Turing Test

Consider the following setting. There are two rooms, A and B. One of the rooms contains a computer. The other contains a human. The interrogator is outside and does not know which one is a computer. He can ask questions through a teletype and receives answers from both A and B. The interrogator needs to identify whether A or B are humans. To pass the Turing test, the machine has to fool the interrogator into believing that it is human. For more details on the Turing test visit the site http://cogsci.ucsd.edu/~asaygin/tt/ttest.html



1.4 LIMITS OF AI TODAY:

Today's successful AI systems operate in well-defined domains and employ narrow, specialized knowledge. Common sense knowledge is needed to function in complex, open-ended worlds. Such a system also needs to understand unconstrained natural language. However these capabilities are not yet fully present in today's intelligent systems.

What can AI systems do?

Today's AI systems have been able to achieve limited success in some of these tasks.

- In Computer vision, the systems are capable of face recognition
- In Robotics, we have been able to make vehicles that are mostly autonomous.
- In Natural language processing, we have systems that are capable of simple machine translation.
- Today's Expert systems can carry out medical diagnosis in a narrow domain
- Speech understanding systems are capable of recognizing several thousand words continuous speech
- Planning and scheduling systems had been employed in scheduling experiments with the Hubble Telescope.

What can AI systems NOT do yet?

- Understand natural language robustly (e.g., read and understand articles in a newspaper)
- Surf the web

• Interpret an arbitrary visual scene
• Learn a natural language
• Construct plans in dynamic real-time domains
• Exhibit true autonomy and intelligence

1.5 APPLICATIONS OF ARTIFICIAL INTELLIGENCE:

Game Playing

- Much of the early research in state space search was done using common board games such as checkers, chess, and the 15-puzzle.
- Games can generate extremely large search spaces. These are large and complex enough to require powerful techniques for determining what alternative to explore.

Automated reasoning and Theorem Proving

- Theorem-proving is one of the most fruitful branches of the field
- Theorem-proving research was responsible in formalizing search algorithms and developing formal representation languages such as predicate calculus and the logic programming language.

Expert System

- One major insight gained from early work in problem solving was the importance of domain-specific knowledge.
- Expert knowledge is a combination of a theoretical understanding of the problem and a collection of heuristic problem-solving rules.

Current deficiencies:

- Lack of flexibility; if human cannot answer a question immediately, he can return to an examination of first principle and come up something.
- Inability to provide deep explanations.
- Little learning from experience.

Natural Language Understanding and Semantics:

- One of the long-standing goals of AI is the creation of programming that are capable of understanding and generating human language.

Modeling Human Performance:

- Capture the human mind (knowledge representation)

Robotics:

- A robot that blindly performs a sequence of actions without responding to changes or being able to detect and correct errors could hardly considered intelligent.
- It should have some degree of sensors and algorithms to guild it.

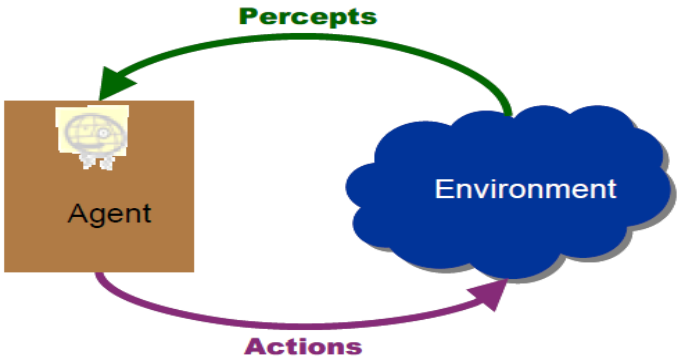
Machine Learning:

- Learning has remained a challenging area in AI.
- An expert system may perform extensive and costly computation to solve a problem; unlike human, it usually don't remember the solution.

1.6 SEARCH IN STATE SPACES:

1.6.1 Agents That Plan:

An agent acts in an environment.



An agent perceives its environment through sensors. The complete set of inputs at a given time is called a percept. The current percept, or a sequence of percepts can influence the actions of an agent. The agent can change the environment through actuators or effectors. An operation involving an effector is called an action. Actions can be grouped into action sequences. The agent can have goals which it tries to achieve.

Thus, an agent can be looked upon as a system that implements a mapping from percept sequences to actions.

A performance measure has to be used in order to evaluate an agent.

An autonomous agent decides autonomously which action to take in the current situation to maximize progress towards its goals.

1.6.2 State space search:

• Formulate a problem as a state space search by showing the legal problem states, the legal operators, and the initial and goal states.

• A state is defined by the specification of the values of all attributes of interest in the world.

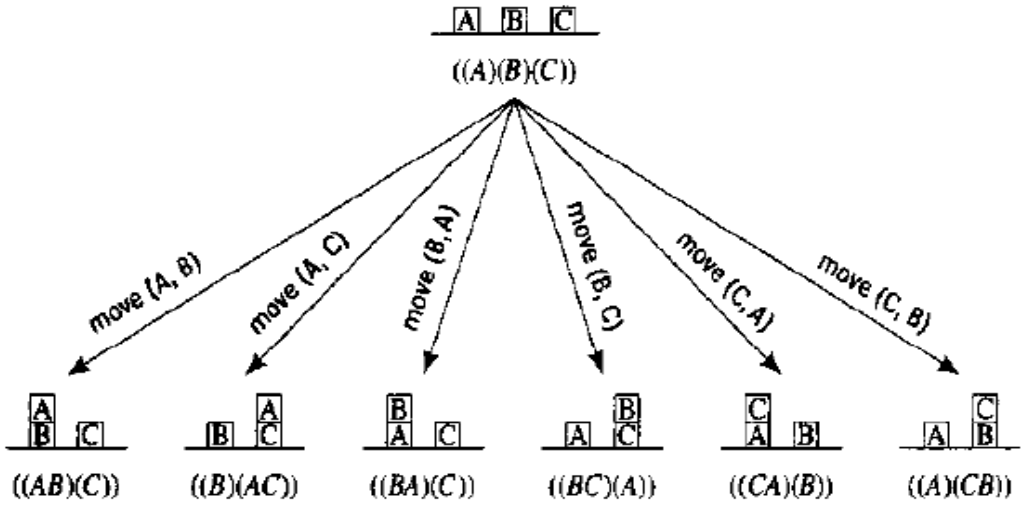
• An operator changes one state into the other; it has a precondition which is the value of certain attributes prior to the application of the operator, and a set of effects, which are the attributes altered by the operator.

• The initial state is where you start.

• The goal state is the partial description of the solution.

1.6.3 State Space Graph:

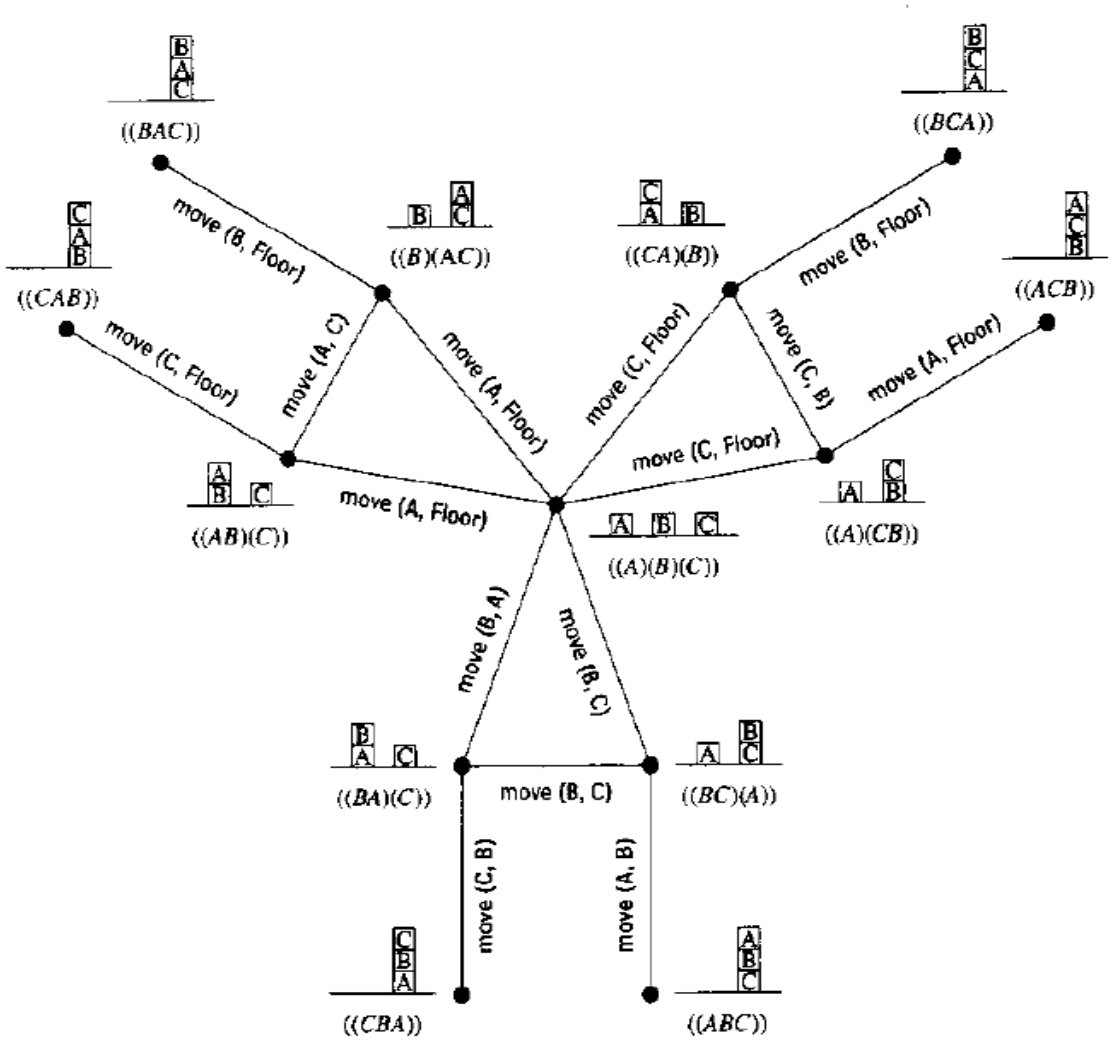
- An example, consider a grid space world containing three toy blocks; A, B, C all initially on the floor. Suppose the task for our robot is to stack them so that A is on top of B and is on top of C and C is on floor.
- While it is obvious for us what actions should be performed, it is not so obvious for a robot.
- A graph representing all of the possible actions and situations.
- Any path in the graph can be taken to represent a goal situation.
- A state space graph is most useful structure for keeping track of the effects of several alternative sequence of actions.
- The nodes in the graph represent possible states of the world and it can be either in iconic dorm or feature form. Each arc is labeled with operators.
- Instances of the schema, such as move (A, C) are called "operators".
- Using list structure iconic models, the modeled effects of all those actions that can be taken when all of the blocks are on the floor.
- Following figure shows the Effects of moving a block:



- Two of these effects, namely ((AB)(C)) and ((A)(BC)), seem to be "closer" in some respects to the goal ((ABC)) then do others.
- The computational agents use several graph searching process to find the path in graph.
- A sequence of operators which are labeled along the path to a goal is called a "plan".
- The process of searching such a plan is called "planning".

- The process of predicting a sequence of world states resulting from a sequence of actions is called "projecting".
- If the number of different distinguishable world situations is sufficiently small, a graph representing all of the possible actions and situations can be stored explicitly.
- For example, the graph illustrated in below figure: shows all the situations and moves relevant to manipulating three blocks.
- It is easy to see from the graph that if the initial situation is given by the world model ((A)(B)(C)), and if the task is to achieve the situation described by ((ABC)), then the robot should execute the following sequence of actions:

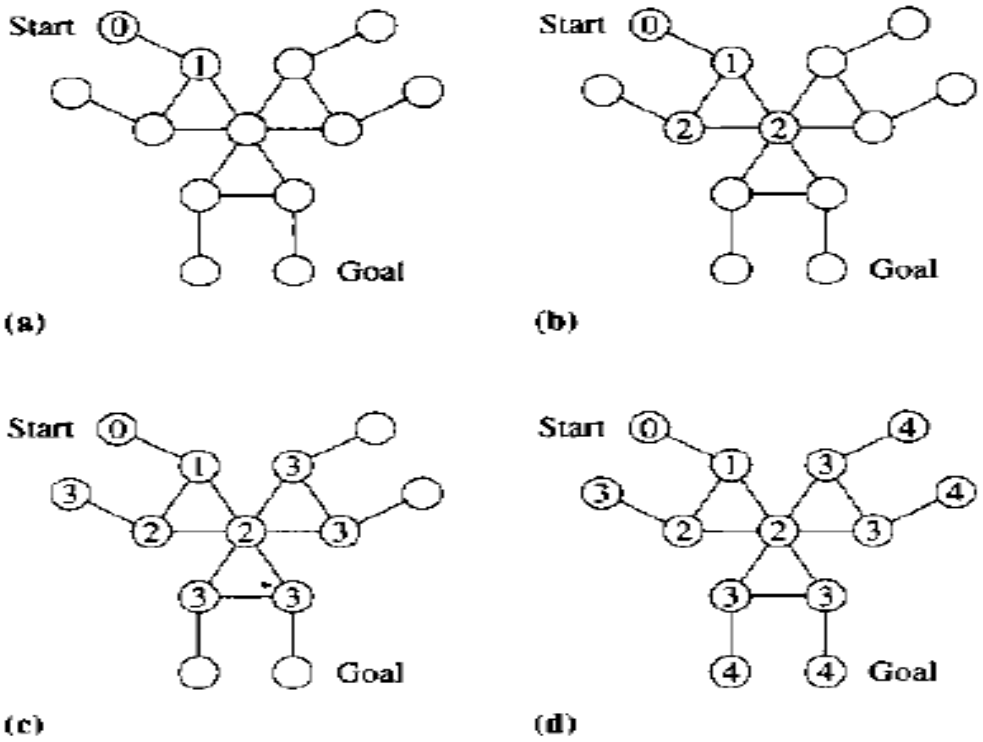
  {move (B, C), move (A, B)}



- One of the advantages of representing the possible worlds in a graph structure is that any of the nodes in the graph can be taken to represent a goal situation.

1.6.4 Searching Explicit State Spaces:

- Search methods for explicit graphs involve propagating "markers" over the nodes of the graph.
- We start by labeling the start node with a 0 and then we propagate successively larger integers out in waves along the arcs until an integer hits the goal.
- Then, we trace a path back from the goal to the start along a decreasing sequence of numbers.
- The actions along this path, from start to goal, are the actions that should be taken to achieve the goal.
- This method requires O(n) steps, where 'n' is the number of nodes in the graph.

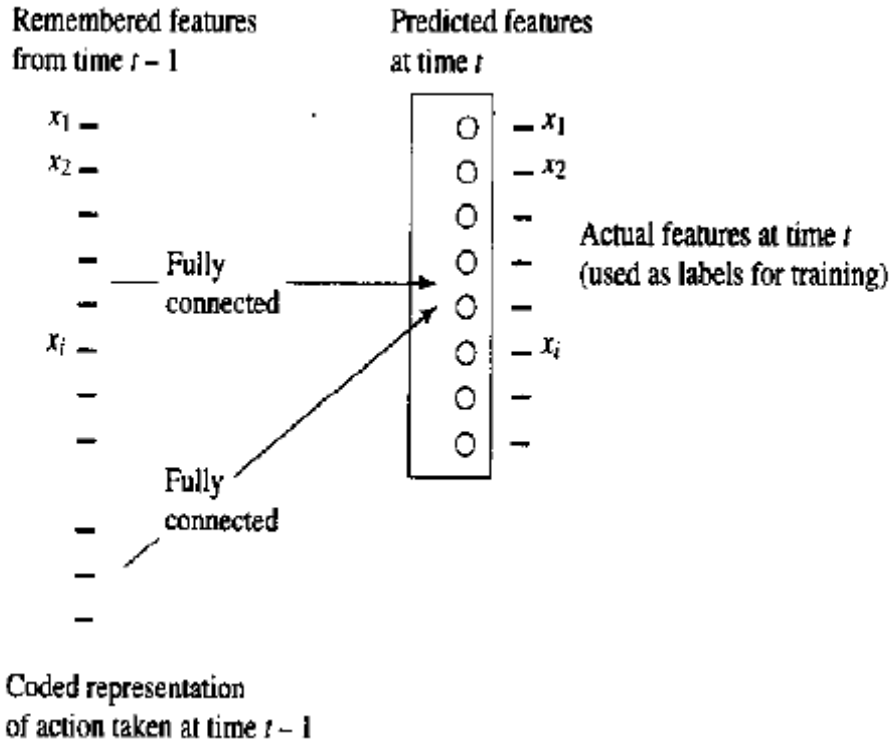


- Breadth first search method is employed to find a sequence across the graph.
- In this method each node is expanded in such a way that marked node puts the next higher integer on all of its unmarked nodes.
- The process repeats until an integer hits the goal.

1.7 FEATURE BASED STATE SPACE:

- It was rather straightforward to explain state spaces when using iconic models to label the nodes.
- We could easily visualize the effects of actions on the states.
- It is also possible to define graphs whose nodes are labeled by features.

- One technique for describing such effects was introduced in a system called STRIPS.
- The basic idea is to define an operator by three lists:
- The first called preconditioned list: specifies those features that must have value 1 and those must have value 0 in order that the action can be applied at all.
- The second called the delete list: specifies those features that will have their values changed from 1 to 0.
- The third called the add list: specifies those features that will have their values changed from 0 to 1.
- The values of the features not explicitly mentioned in the delete and add lists are unchanged.
- These three lists compose a STRIPS operator-a model of the effects of an action.
- We might also be able to train a neural network to learn to predict the value of a feature vector at time t from its value at time t-1 and the action taken at time t-1. Such network is shown in below figure:



Predicting a Feature Vector with a Neural Network

1.8 GRAPH NOTATION:

- It will be helpful to define some of the accepted terminology that is used in discussions of graph search. The below figure shows graph and tree notation.

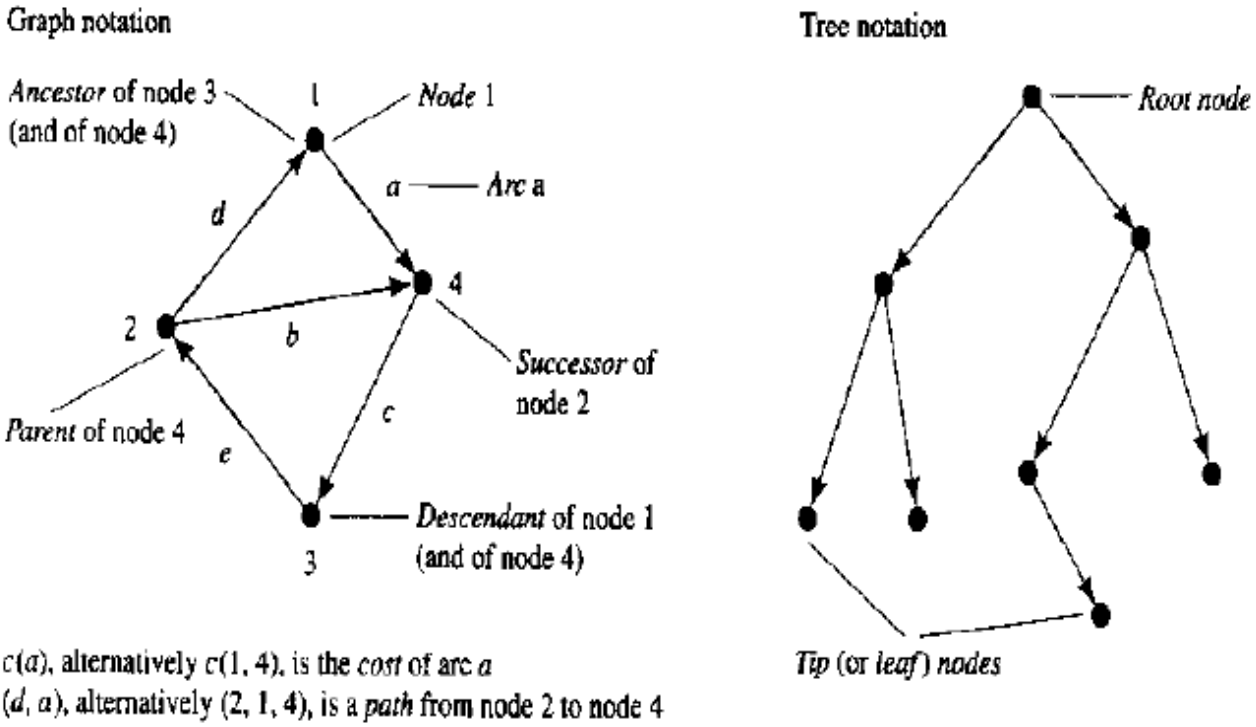


Figure: Graph and Tree Notation

- A Graph consists of set of nodes. Certain pair of nodes connected by arcs these arcs are directed from one member of pair to the other.
- Such a graph is called directed graph.
- If an arc is directed from node n_i to node n_j then node n_j is said to be successor of node n_i and node n_i is said to be parent of node n_j.
- A node can have only a finite number of successors.
- A pair of nodes can be successors of each other; in this case we replace the pair of an arc by edge.
- Graphs that contain only edges called undirected graph.
- A directed tree is a special case of directed graph in which each node has exactly one parent.
- The node with no parent is called the root node.
- A node in the tree having no successors is called a tip node or leaf node.
- We say that the root node is of depth zero. The depth of any other node in the tree is defined to be the depth of its parent plus 1.

- A (rooted) undirected tree(with edges instead of arcs) is an undirected graph in which there is precisely one and only one path along edges between any pair of nodes.
- Certain trees, useful in theoretical analyses, have the property that all nodes except the tip nodes have the same number, b, of successors. In this case 'b' is called the branching factor of the tree.

1.9 UNINFORMED SEARCH:

- It is also called ad blind search.
- The uninformed search strategies are given no information about the problem other than its definition.
- It simply generate successor and distinguishes a goal state from a non goal state.

1.9.1 Formulating the State Space:

- Many problems of practical interest have search spaces so large that they cannot be represented by explicit graphs. First, we need to be especially careful about how we formulate such problems for search; second, we have to have methods for representing large search graphs explicitly; and third, we need to use efficient methods for searching such large graphs.
- A typical example is the fifteen-puzzle, which consists of fifteen tiles set in a four-by four array-leaving one empty or blank cell into which an adjacent tile can be slide.
- The task is to find a sequence of tile movements that transforms an initial disposition of tiles into some particular arrangement.
- The eight-puzzle is a reduced version with eight tiles in a three-by-three array.
- Suppose the object of the puzzle is to slide tiles from the starting configuration until the goal configuration is reached, as shown in below figure:

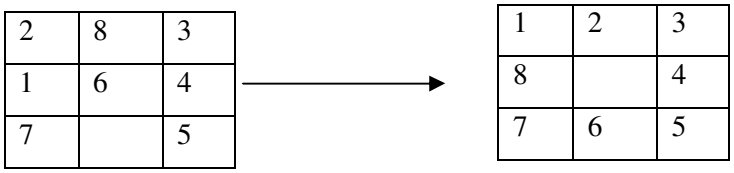


Figure: start and goal configurations for eight puzzle:

1.9.2 Components of Implicit State-Space Graphs:

There are three basic components to an implicit representation of a state space graph:

1) A description with which to label the start node. This description is some data structure modeling the initial state of the environment.
2) Functions that transform a state description representing one state of the environment into one that represents the state resulting after an action. These functions usually called operators.

3) A goal condition, which can be either a true-false-valued function on state description or a list of actual instance of state descriptions that correspond to goal states.

1.9.3Breadth First Search:

- Uninformed search procedures apply operators to nodes without using any special knowledge about the problem domain. Perhaps the simplest uninformed search procedure is breadth-first search. That procedure generates an explicit state- space graph by applying all possible operators to the start node, then applying all possible operators to all the directed successors of the start node, then to their successors, and so on.
- Search proceeds uniformly outward from the start node. Since we apply all possible operators to a node at each step, it is convenient to group them into a function called the successor function.
- The successor function, when applied to a node, produces the entire set of nodes that can be produced by applying all of the operators that can be applied to that node.
- Each application of a successor function to a node is called expanding the node.
- Below figure shows the nodes created in a breadth-first search for a solution to the eight-puzzle problem.
- The start node and the goal nodes are labeled, and the order of node expansions is shown by a numeral next to each node.
- Nodes of the same depth are expanded according to some fixed order.
- In expanding a node operators are in the order: move left, up, right, down. Even though is move is reversible.
- The solution path is shown by the dark line. As we se, breadth-first search has he property that when a goal node is found, we have found a path of minimal length to the goal.
- A disadvantage of breadth-first search, however, is that it requires the generation and the storage of a tree whose size is exponential in the depth of the shallowest goal node.

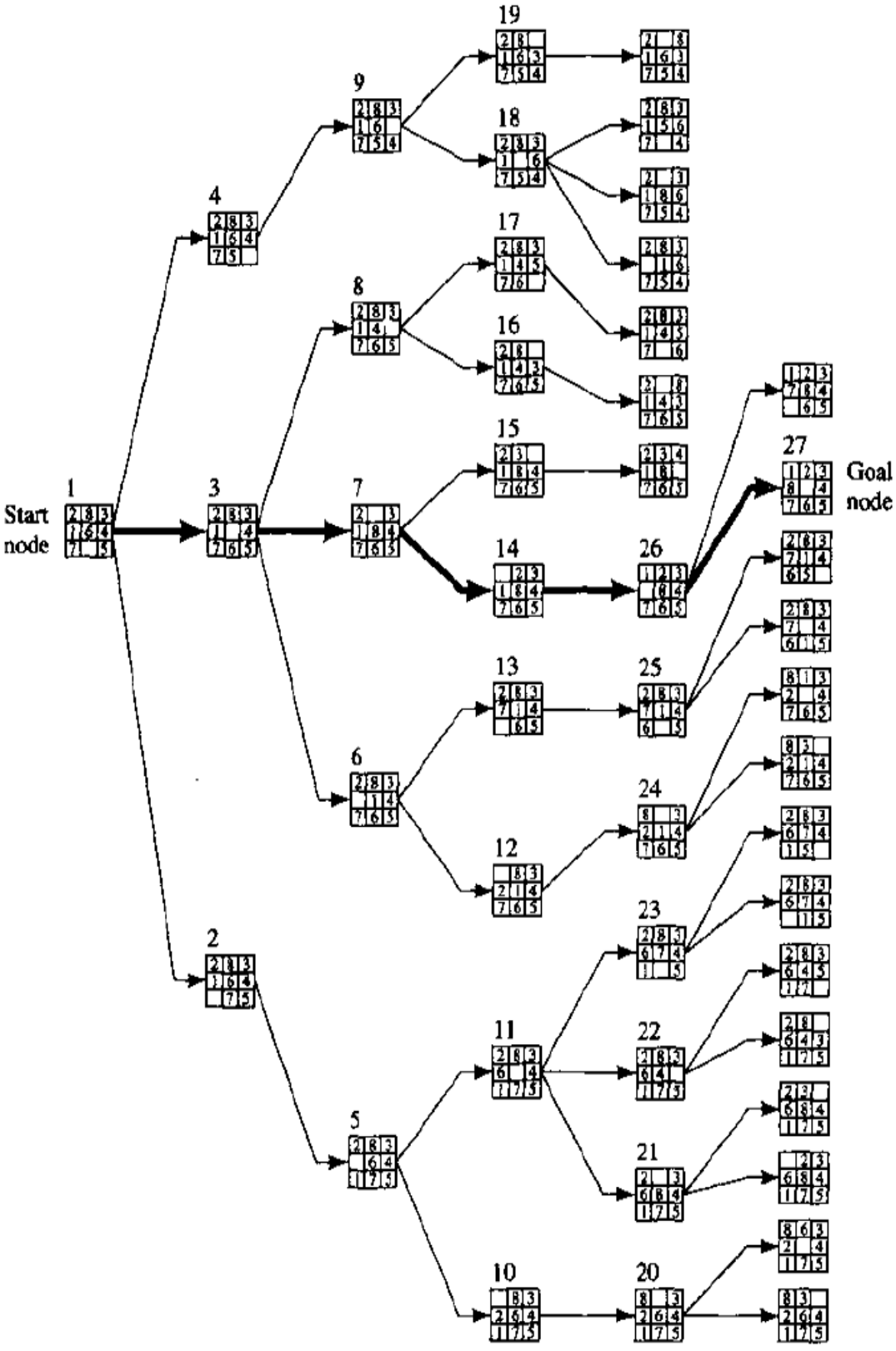
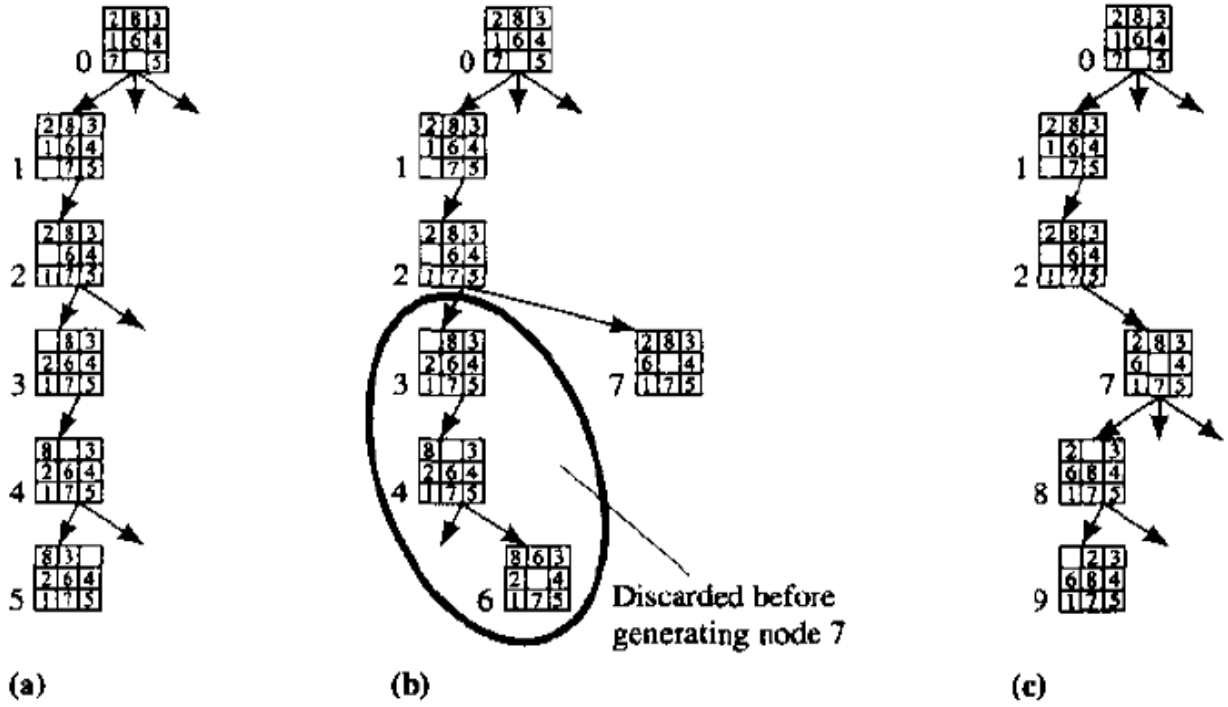


Figure: breadth first search of the eight puzzle

1.9.4 Depth First Search or Backtracking Tracking Search:

- Depth-first search generates the successors of a node just one at a time by applying individual operators.
- A trace is left at each node to indicate that additional operators can eventually be applied there if needed
- At each node, a decision must be made about which operator to apply first, which next, and so on.
- As soon as a successor is generated, one of its successors is generated and so on.
- To prevent the search process from running away toward nodes of unbounded depth from the start node, a depth bound is used.
- No successor is generated whose depth is greater than the depth bound.
- This bound allows us to ignore parts of the search graph that have been determined not to contain a sufficiently close goal node.
- We apply operators in the order move blank left, up, down and we omit arcs from successors back to their parents.
- Below figure (a) shows the first few node generations.
- The number at the left of each node shows the order in which the node is generated. At node 5, we reach the depth bound without having reached the goal, so we consider the next most recently generated, but not yet fully expanded node, node 4 and generate the order of its successors, node6(see figure b).figure: generation of the first few nodes in the depth-first search:



- At this point, we can throw away node 5, since we are not going to generate nodes below it.
- Node 6 is also at the depth bound and is not a goal node, so we consider the next most recently generated but not fully expanded node, node2 and generate node 7 throwing away node3 and its successors since we are not going to generate any more nodes below them.
- Going back to node 2 is an example of what is called chronological backtracking. When the depth bound is reached again, we have the graph shown in figure c.
- At this point not having reached a goal node, we generate the other successor of node 8, which is not a goal node either.
- So, we throw away node 8 and its successor and backtrack to generate another successor of node7. Continues this process finally results in the below shown graph:

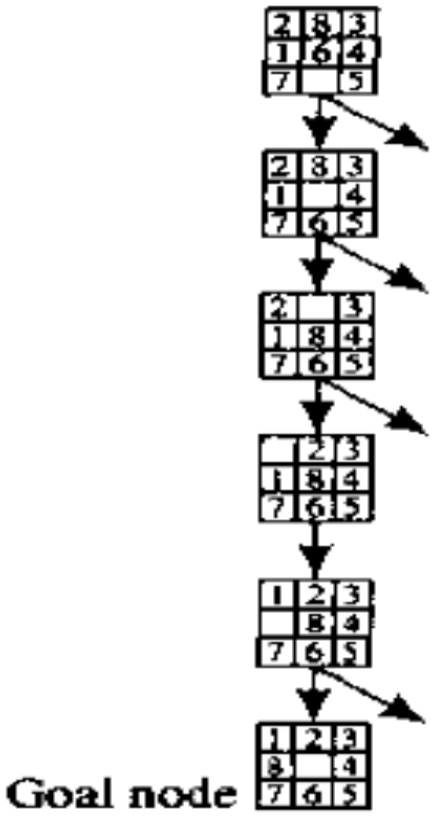


Figure: the graph when the goal is reached in depth first search

1.10 HEURISTIC SEARCH

The search processes that I describe in this chapter are something like breadth-first search, except that search does not proceed uniformly outward from the start node; instead, it proceeds preferentially through nodes that heuristic, problem-specific information indicates might be on the best path to a goal. We call such processes *best-first* or *heuristic* search. Here is the basic idea.

1. We assume that we have a heuristic (evaluation) function, $\hat{f}$, to help decide which node is the best one to expand next. (The reason for the "hat" over the f will become apparent later. We pronounce $\hat{f}$ "f-hat.") We adopt the convention that small values of $\hat{f}$ indicate the best nodes. This function is based on information specific to the problem domain. It is a real-valued function of state descriptions.
2. Expand next that node, n, having the smallest value of $\hat{f}(n)$. Resolve ties arbitrarily. (I assume in this chapter that node expansion produces all of the successors of a node.)
3. Terminate when the node to be expanded next is a goal node.

People are often able to specify good evaluation functions for best-first search. In the Eight-puzzle, for example, we might try using the number of tiles out of place as a measure of the goodness of a state description:

$\hat{f}(n)$ = number of tiles out of place (compared with goal)

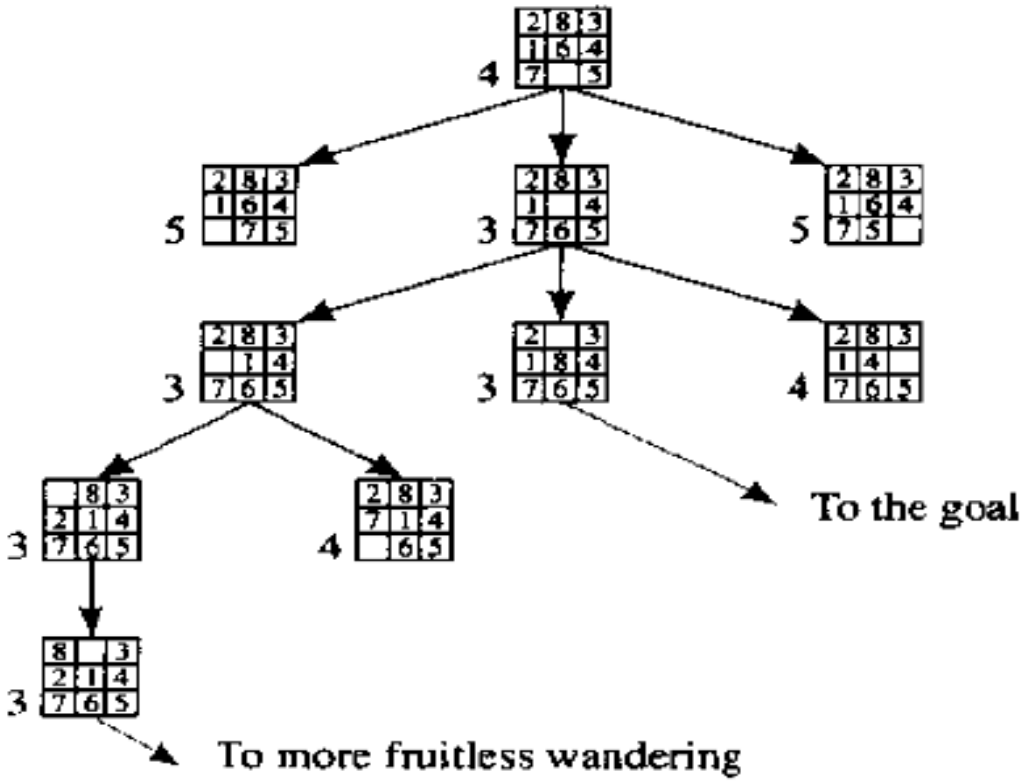


Figure 9.1

Using this heuristic function in the search procedure just described produces the graph shown in Figure 9.1. The numeral next to each node is the value of $\hat{f}$ for that node.

This example shows that we need to bias the search in favor of going back to explore early paths (in order to prevent being led "down a garden path" by an overly optimistic heuristic). So, we add a "depth factor" to $\hat{f}$: $\hat{f}(n) = \hat{g}(n) + \hat{h}(n)$, where $\hat{g}(n)$ is an estimate of the "depth" of n in the graph (that is, the length of the shortest path from the start to n), and $\hat{h}(n)$ is a heuristic evaluation of node n. If, as before, we let $\hat{h}(n)$ = number of tiles out of place (compared with the goal) and take $\hat{g}(n)$ = the depth of node n in the search graph, we get the graph shown in Figure 9.2. In this figure, I have written the values of $\hat{g}(n) + \hat{h}(n)$ next to each node. We see that search proceeds rather directly toward the goal in this case (with the exception of the circled node).

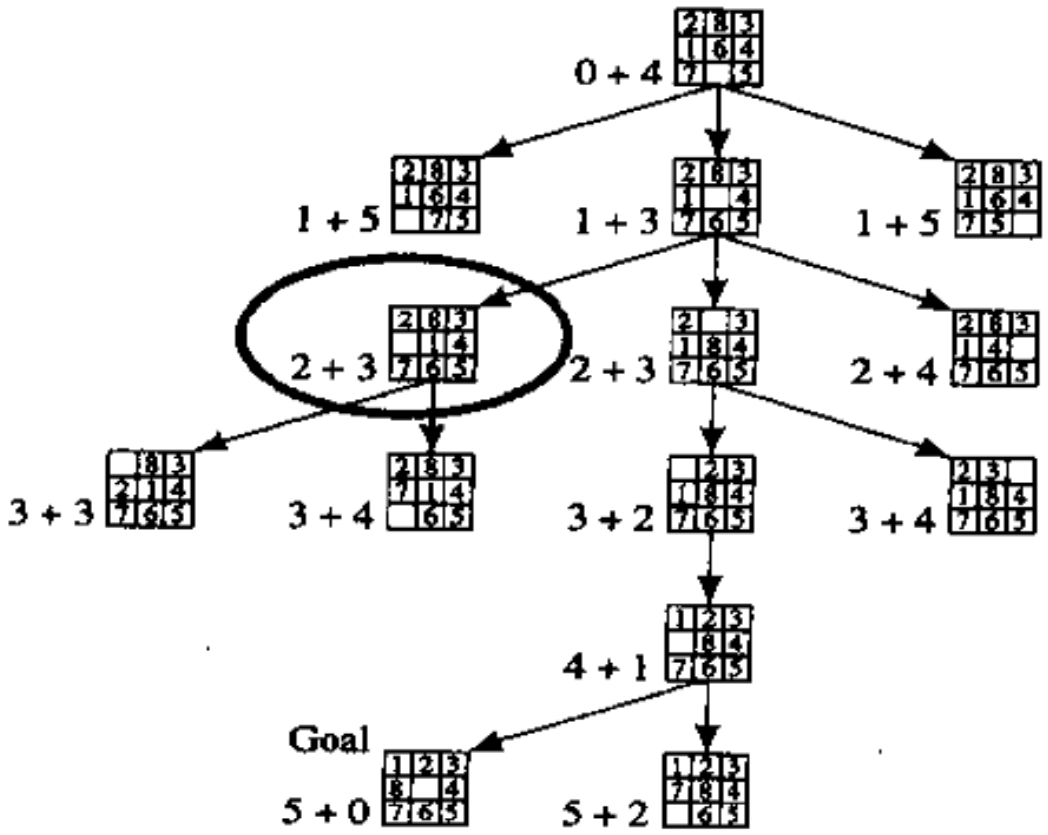


Figure 9.2

Heuristic Search Using $\hat{f}(n) = \hat{g}(n) + \hat{h}(n)$

1.10.1 A General Graph Searching Algorithm:

GRAPHSEARCH

1. Create a search tree, *Tr*, consisting solely of the start node, n_0. Put n_0 on an ordered list called *OPEN*.
2. Create a list called *CLOSED* that is initially empty.
3. If *OPEN* is empty, exit with failure.
4. Select the first node on *OPEN*, remove it from *OPEN*, and put it on *CLOSED*. Call this node n.
5. If n is a goal node, exit successfully with the solution obtained by tracing a path backward along the arcs in *Tr* from n to n_0. (Arcs are created in step 6.)

6. Expand node n, generating a set, $\mathcal{M}$, of successors. Install $\mathcal{M}$ as successors of n in Tr by creating arcs from n to each member of $\mathcal{M}$.
7. Reorder the list *OPEN*, either according to some arbitrary scheme or according to heuristic merit.
8. Go to step 3.

This algorithm can be used to perform best-first search, breadth-first search, or depth-first search. In breadth-first search, new nodes are simply put at the end of *OPEN* (first in, first out, or FIFO), and the nodes are not reordered. In depth-first-style search, new nodes are put at the beginning of *OPEN* (last in, first out, or LIFO). In best-first (also called heuristic) search, *OPEN* is reordered according to the heuristic merit of the nodes.

1.10.2 Algorithm A*:

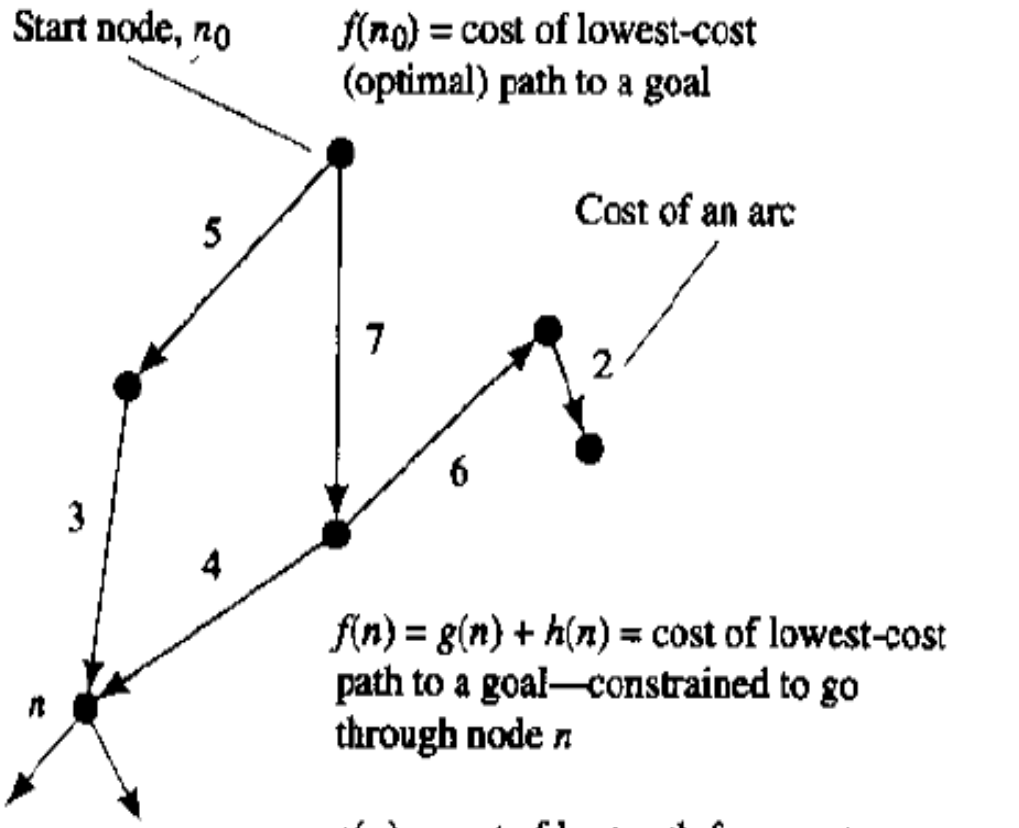
I will particularize GRAPHSEARCH to a best-first search algorithm that reorders (in step 7) the nodes on *OPEN* according to increasing values of a function $\hat{f}$, as in the Eight-puzzle illustration. I will call this version of GRAPHSEARCH, algorithm A*. As we shall see, it will be possible to define $\hat{f}$ functions that will make A* perform either breadth-first or uniform-cost search. In order to specify the family of $\hat{f}$ functions to be used, I must first introduce some additional notation.

Let $h(n)$ = the *actual* cost of the minimal cost path between node n and a goal node (over all possible goal nodes and over all possible paths from n to them).

Let $g(n)$ = the cost of a minimal cost path from the start node, n_0, to node n.

Then $f(n) = g(n) + h(n)$ is the cost of a minimal cost path from n_0 to a goal node over all paths that are constrained to go through node n. Note that $f(n_0) = h(n_0)$ is then the cost of an (unconstrained) minimal cost path from n_0 to a goal node.

For each node, n, let $\hat{h}(n)$ (the heuristic factor) be some estimate of $h(n)$, and let $\hat{g}(n)$ (the depth factor) be the cost of the lowest-cost path found by A* so far to node n. In algorithm A* we use $\hat{f} = \hat{g} + \hat{h}$. Note that algorithm A* with $\hat{h}$ identically 0 gives uniform-cost search. These definitions are illustrated in Figure 9.3.



$\hat{f}$, $\hat{g}$, $\hat{h}$ are estimates of f, g, h, respectively

$\hat{f} = \hat{g} + \hat{h}$

Figure 9.3

Figure: Heuristic search notation

To prevent duplicated search effort when these yet-to-be-mentioned conditions on $\hat{f}$ do not prevail requires some modification to algorithm A*. Because search might reach the same node along different paths, algorithm A* generates a search graph, which we call *G*. *G* is the structure of nodes and arcs generated by A* as it expands the start node, its successors, and so on. A* also maintains a search tree, *Tr*. *Tr*, a subgraph of *G*, is the tree of best (minimal cost) paths produced so far to all of the nodes in the search graph. I show examples for a graph having unit arc costs in Figure 9.4. An early stage of search is shown in Figure 9.4a. The search-tree part of the search graph is indicated by the dark arcs; gray arcs are in the search graph but not the search tree. The dark arcs indicate the least costly paths found so far to the nodes in the search graph. Note that node 4 in Figure 9.4a has been reached by two paths; both paths are in the search graph, but only one is in the search tree. We keep the search graph because subsequent search may find shorter paths that use some of the arcs in the earlier search graph that were not in the earlier search tree. For example, in Figure 9.4b, expanding node 1 finds shorter paths to node 2 and its descendants (including node 4), so the search tree is changed accordingly.

For completeness, I state the version of A* that maintains the search graph. We note, however, that this version is seldom needed because we can usually impose conditions on $\hat{f}$ that guarantee that when algorithm A* expands a node, it has already found the least costly path to that node.

Algorithm A*

1. Create a search graph, *G*, consisting solely of the start node, n_0. Put n_0 on a list called *OPEN*.
2. Create a list called *CLOSED* that is initially empty.
3. If *OPEN* is empty, exit with failure.
4. Select the first node on *OPEN*, remove it from *OPEN*, and put it on *CLOSED*. Call this node *n*.
5. If *n* is a goal node, exit successfully with the solution obtained by tracing a path along the pointers from *n* to n_0 in *G*. (The pointers define a search tree and are established in step 7.)

6. Expand node n, generating the set, $\mathcal{M}$, of its successors that are not already ancestors of n in G. Install these members of $\mathcal{M}$ as successors of n in G.
7. Establish a pointer to n from each of those members of $\mathcal{M}$ that were not already in G (i.e., not already on either *OPEN* or *CLOSED*). Add these members of $\mathcal{M}$ to *OPEN*. For each member, m, of $\mathcal{M}$ that was already on *OPEN* or *CLOSED*, redirect its pointer to n if the best path to m found so far is through n. For each member of $\mathcal{M}$ already on *CLOSED*, redirect the pointers of each of its descendants in G so that they point backward along the best paths found so far to these descendants.
8. Reorder the list *OPEN* in order of increasing $\hat{f}$ values. (Ties among minimal $\hat{f}$ values are resolved in favor of the deepest node in the search tree.)
9. Go to step 3.

In step 7, we redirect pointers from a node if the search process discovers a path to that node having lower cost than the one indicated by the existing pointers. Redirecting pointers of descendants of nodes already on *CLOSED* saves subsequent search effort but at the possible expense of an exponential amount of computation. Hence this part of step 7 is often not implemented. Some of these pointers will ultimately be redirected in any case as the search progresses.

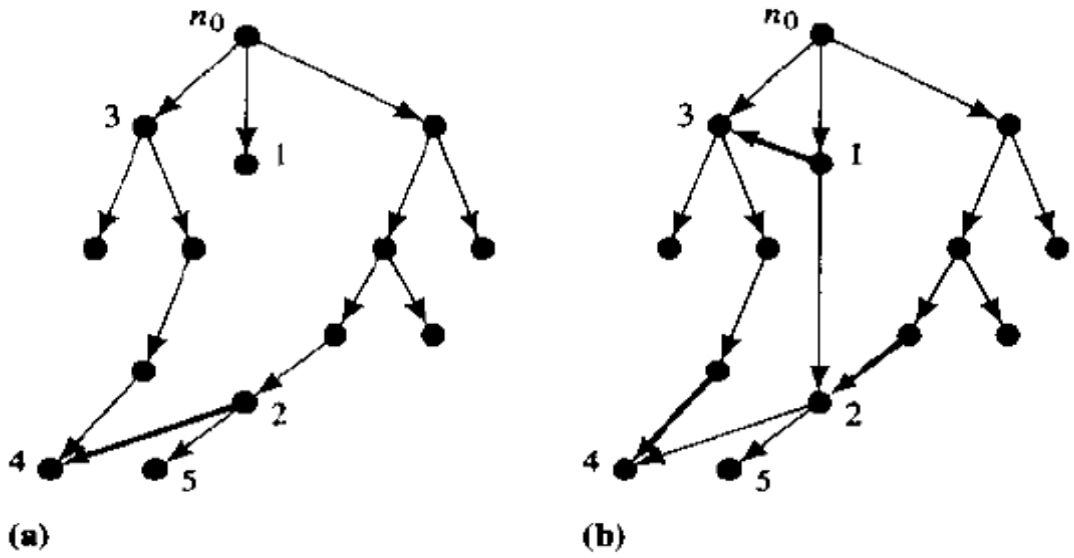
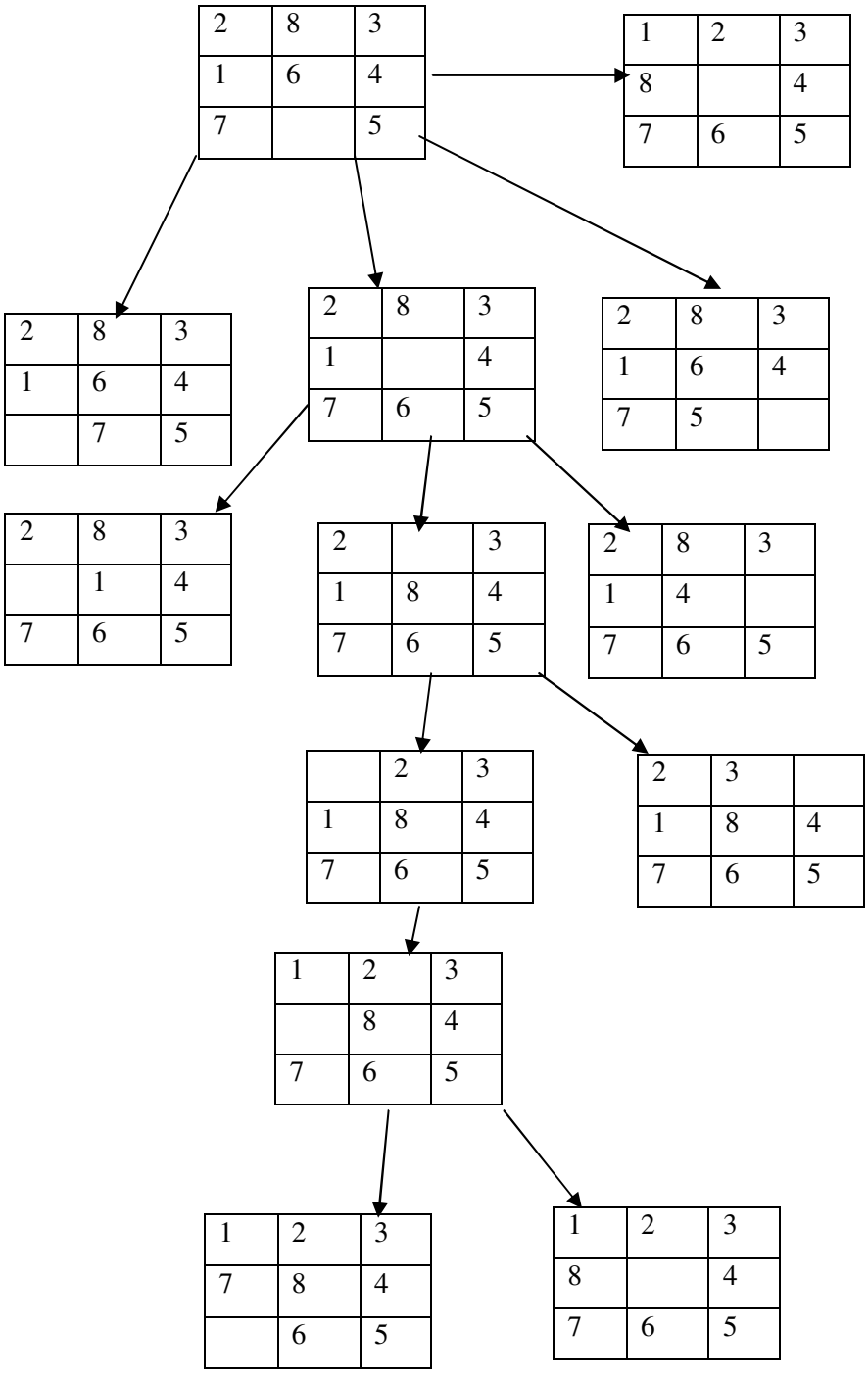


Figure 9.4

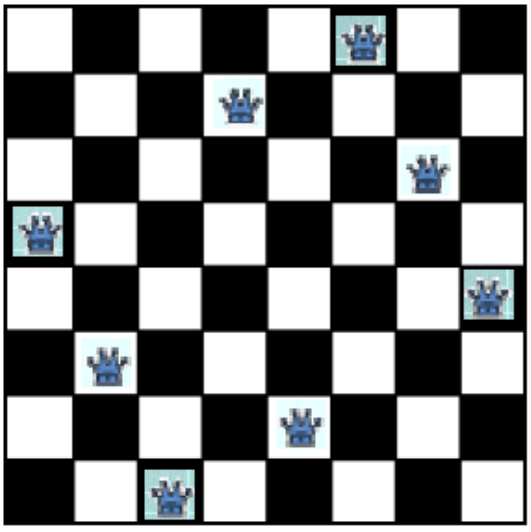
Search Graphs and Trees Produced by a Search Procedure

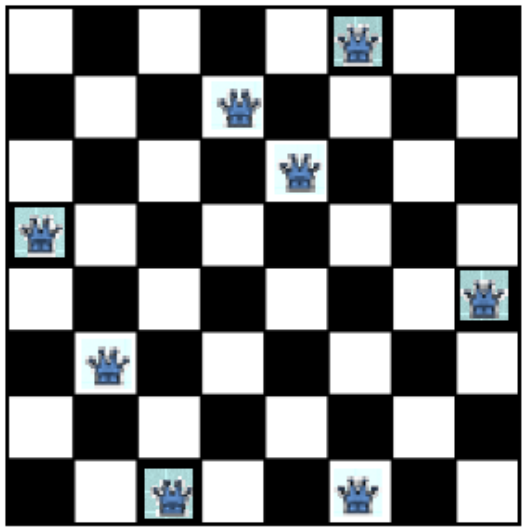
Example: 1) heuristic search applying for eight puzzle problem:



Example2) 8 queen's problem

- The problem is to place 8 queens on a chessboard so that no two queens are in the same row, column or diagonal.
- The picture below on the left shows a solution of the 8-queens problem. The picture on the right is not a correct solution, because some of the queens are attacking each other.



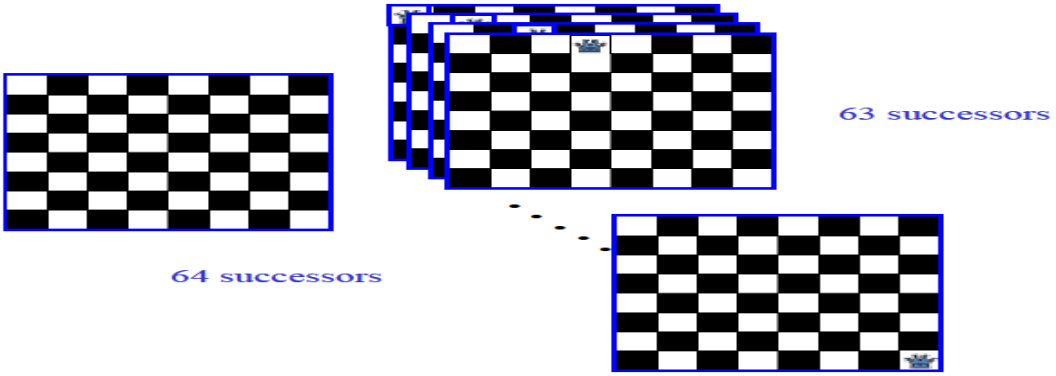


- How do we formulate this in terms of a state space search problem? The problem formulation involves deciding the representation of the states, selecting the initial state representation, the description of the operators, and the successor states. We will now show that we can formulate the search problem in several different ways for this problem.

N queens problem formulation 1

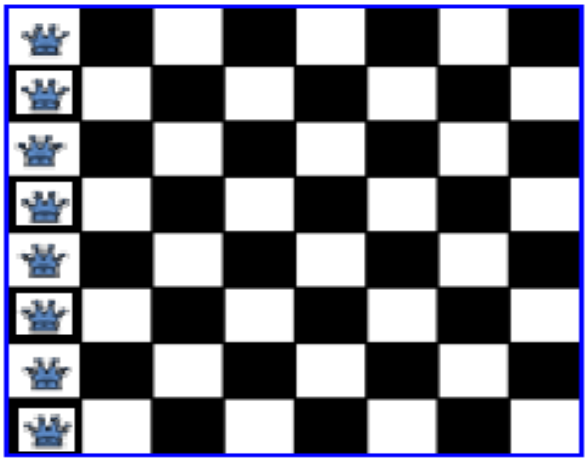
- States: Any arrangement of 0 to 8 queens on the board
- Initial state: 0 queens on the board
- Successor function: Add a queen in any square
- Goal test: 8 queens on the board, none are attacked

The initial state has 64 successors. Each of the states at the next level have 63 successors, and so on. We can restrict the search tree somewhat by considering only those successors where no queen is attacking each other. To do that we have to check the new queen against all existing queens on the board. The solutions are found at a depth of 8.



N queens problem formulation 2

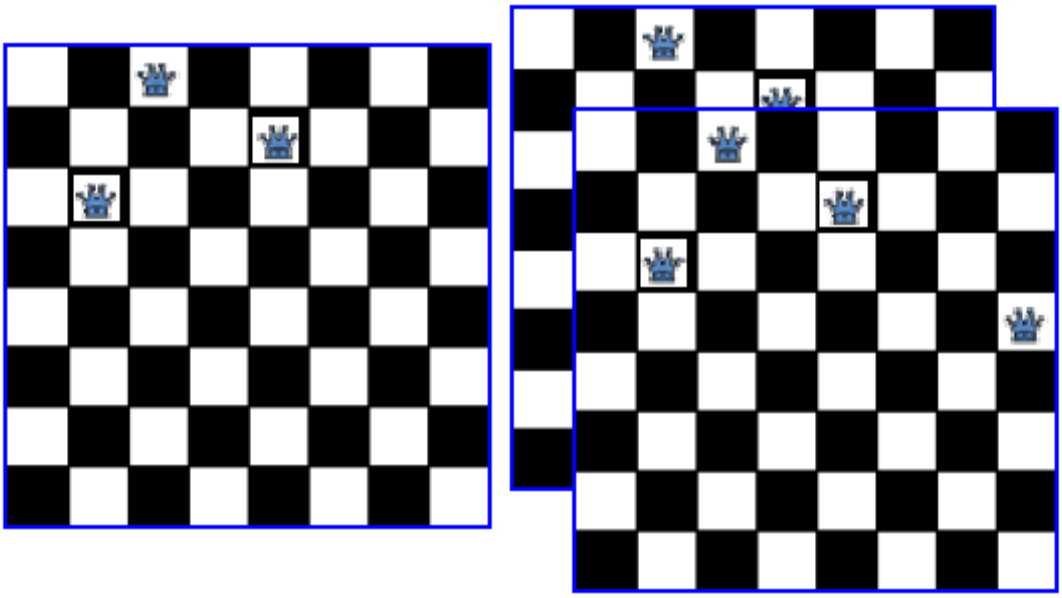
• States: Any arrangement of 8 queens on the board
• Initial state: All queens are at column 1
• Successor function: Change the position of any one queen
• Goal test: 8 queens on the board, none are attacked



If we consider moving the queen at column 1, it may move to any of the seven remaining columns.

N queens problem formulation 3

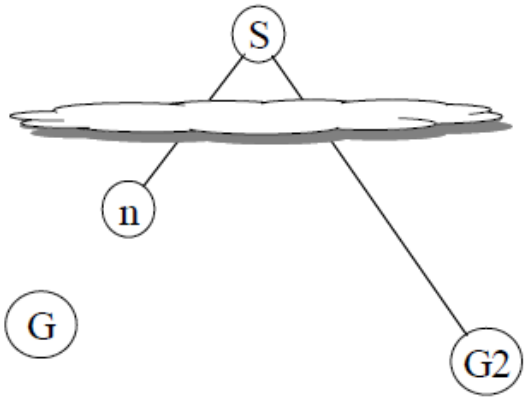
• States: Any arrangement of k queens in the first k rows such that none are attacked
• Initial state: 0 queens on the board
• Successor function: Add a queen to the (k+1)th row so that none are attacked.
• Goal test : 8 queens on the board, none are attacked



1.10.1 Proof of Admissibility of A*:

- We will show that A* is admissible if it uses a monotone heuristic.
- A monotone heuristic is such that along any path the f-cost never decreases.
- But if this property does not hold for a given heuristic function, we can make the f value monotone by making use of the following trick (m is a child of n)
  $f(m) = max\ (f(n),\ g(m) + h(m))$
  - Let G be an optimal goal state
  - C* is the optimal path cost.
  - G2 is a suboptimal goal state: g(G2) > C*

Suppose A* has selected G2 from OPEN for expansion.



- Consider a node n on OPEN on an optimal path to G. Thus $C^* \geq f(n)$
- Since *n* is not chosen for expansion over G2, $f(n) \geq f(G2)$
- G2 is a goal state. $f(G2) = g(G2)$
  Hence $C^* \geq g(G2)$.

This is a contradiction. Thus A* could not have selected G2 for expansion before reaching the goal by an optimal path.

1.10.2 Proof of Completeness of A*:

- Let G be an optimal goal state.
- A* cannot reach a goal state only if there are infinitely many nodes where f(n) ≤ C*.
- This can only happen if either happens:

There is a node with infinite branching factor. The first condition takes care of this.

There is a path with finite cost but infinitely many nodes. But we assumed that every arc in the graph has a cost greater than some ε> 0. Thus if there are infinitely many nodes on a path g(n) > f*, the cost of that path will be infinite.

- Lemma: A* expands nodes in increasing order of their f values.
- A* is thus complete and optimal, assuming an admissible and consistent heuristic function (or using the pathmax equation to simulate consistency).

A* is also optimally efficient, meaning that it expands only the minimal number of nodes needed to ensure optimality and completeness.

1.10.3 Properties of Heuristics

Dominance:
h2 is said to dominate h1 iff $h2(n) \geq h1(n)$ for any node *n*.
A* will expand fewer nodes on average using h2 than h1.

Proof:

Every node for which $f(n) < C^*$ will be expanded. Thus n is expanded whenever

$h(n) < f^* - g(n)$

Since $h2(n) \geq h1(n)$ any node expanded using h2 will be expanded using h1.

1.10.4 Iterative-Deepening A*

IDA* Algorithm

Iterative deepening A* or IDA* is similar to iterative-deepening depth-first, but with the following modifications:
The depth bound modified to be an f-limit

1. Start with limit = h(start)
2. Prune any node if f(node) > f-limit
3. Next f-limit=minimum cost of any node pruned

The cut-off for nodes expanded in an iteration is decided by the f-value of the nodes.

Figure 1 drwa

- Consider the graph in Figure 3. In the first iteration, only node a is expanded. When a is expanded b and e are generated. The f value of both are found to be 15.
- For the next iteration, a f-limit of 15 is selected, and in this iteration, a, b and c are expanded.

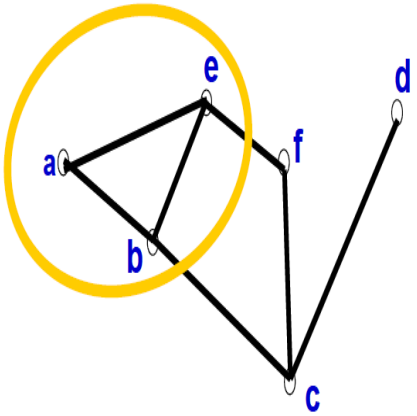


Figure 2: f-limit = 15

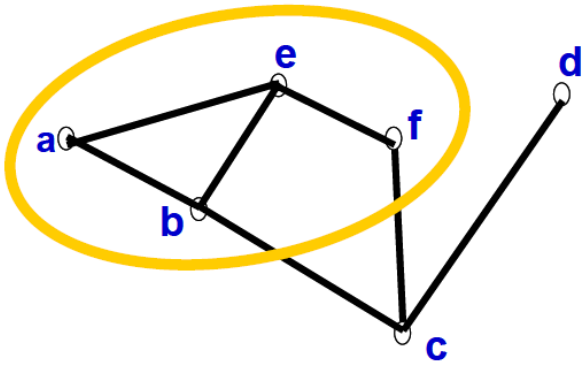


Figure 3: f-limit = 21

RBFS: Recursive Breadth First Search

- RBFS uses only linear space.
- It mimics best first search.
- It keeps track of the f-value of the best alternative path available from any ancestor of the current node.
- If the current node exceeds this limit, the alternative path is explored.

RBFS remembers the f-value of the best leaf in the forgotten sub-tree.

```
RBFS (node: N, value: F(N), bound: B)

   IF f(N)>B, RETURN f(N)
   IF N is a goal, EXIT algorithm
   IF N has no children, RETURN infinity
   FOR each child Ni of N,
         IF f(N)<F(N), F[i] := MAX(F(N),f(Ni))
         ELSE F[i] := f(Ni)
sort Ni and F[i] in increasing order of F[i]
   IF only one child, F[2] := infinity
   WHILE (F[1] <= B and F[1] < infinity)
            F[1] := RBFS(N1, F[1], MIN(B, F[2]))
     insert Ni and F[1] in sorted order
RETURN F[1]
```

1.11 ALTERNATIVE SEARCH FORMULAATIONS AND APPLICATIONS

1.11.1 Heuristic Search:

There is another approach to setting up a problem for solution by graph-search methods. It is called the *repair approach* because it starts with a proposed solution, which most probably does not satisfy the constraints, and repairs it until it does. Thus, the initial node is labeled by a data structure that typically does not satisfy all of the constraints. The operators produce a new data structure that corresponds to a different proposed solution.

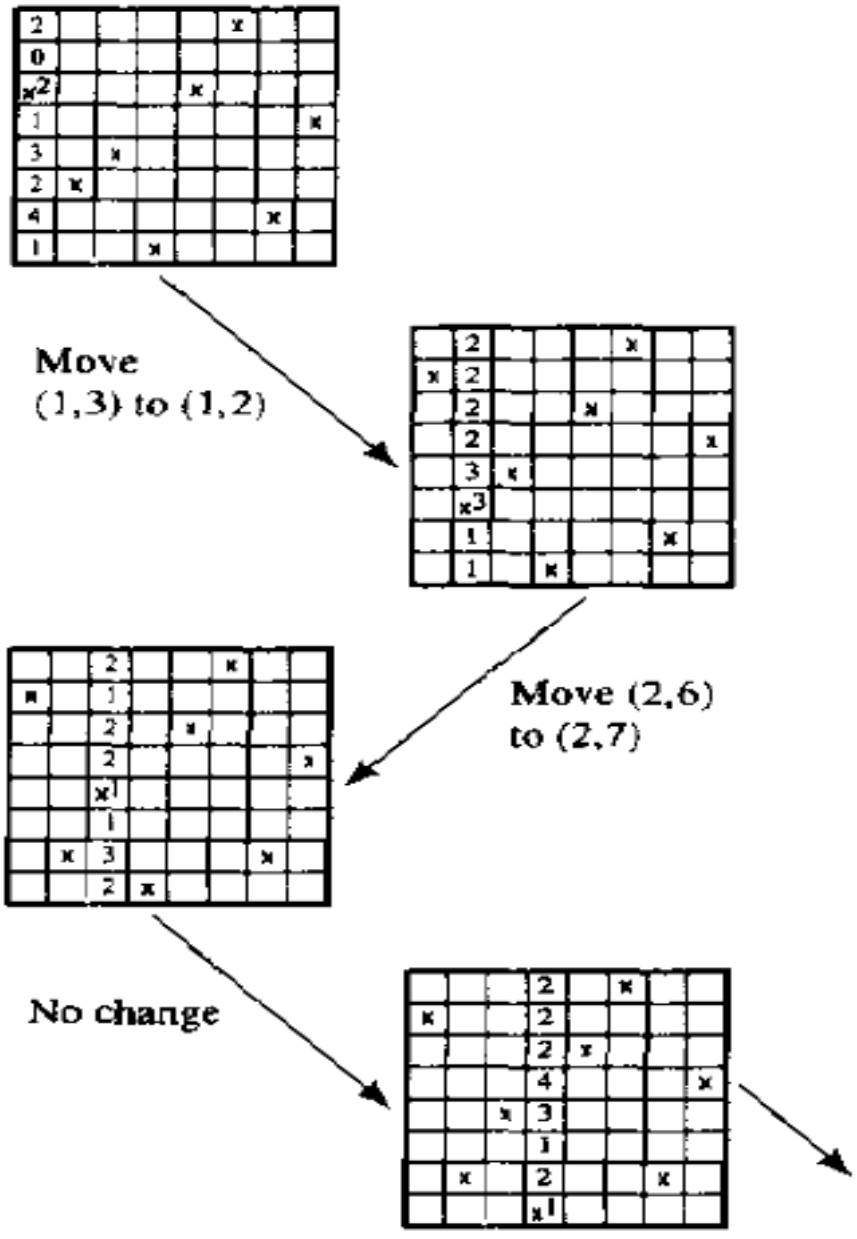


Figure 11.7
Repair Steps for the Eight-Queens Problem

For example, in the Eight-Queens problem, we might start with eight queens, one in each column, and in arbitrary, perhaps random, row positions. We repair a faulty solution by moving one of the queens to violate fewer constraints. In the so-called *min-conflicts* [Gu 1989, Minton, et al. 1992] version of repair, we consider each column in turn (starting, say, with the first) and label each cell in that column with the number of queens (outside of that column) that attack that cell. Then we move the queen in that column to the cell having the minimum number of attacking queens (the minimum number of conflicts). Ties are broken randomly. This operation produces a successor node labeled by (perhaps) a slightly repaired proposed solution. And so on through the columns. I illustrate part of a depth-first search for the Eight-Queens problem, using min-conflicts, in Figure 11.7. Again, queen locations are indicated by X's, and the numbers in the cells refer to the number of queens attacking that cell. Similar repair-based approaches [Minton, et al. 1990] have been used to solve much larger problems, such as the Million-Queens problem.[2]

1.12 Adversarial Search:

We will set up a framework for formulating a multi-person game as a search problem. We will consider games in which the players alternate making moves and try respectively to maximize and minimize a scoring function (also called utility function). To simplify things a bit, we will only consider games with the following two properties:

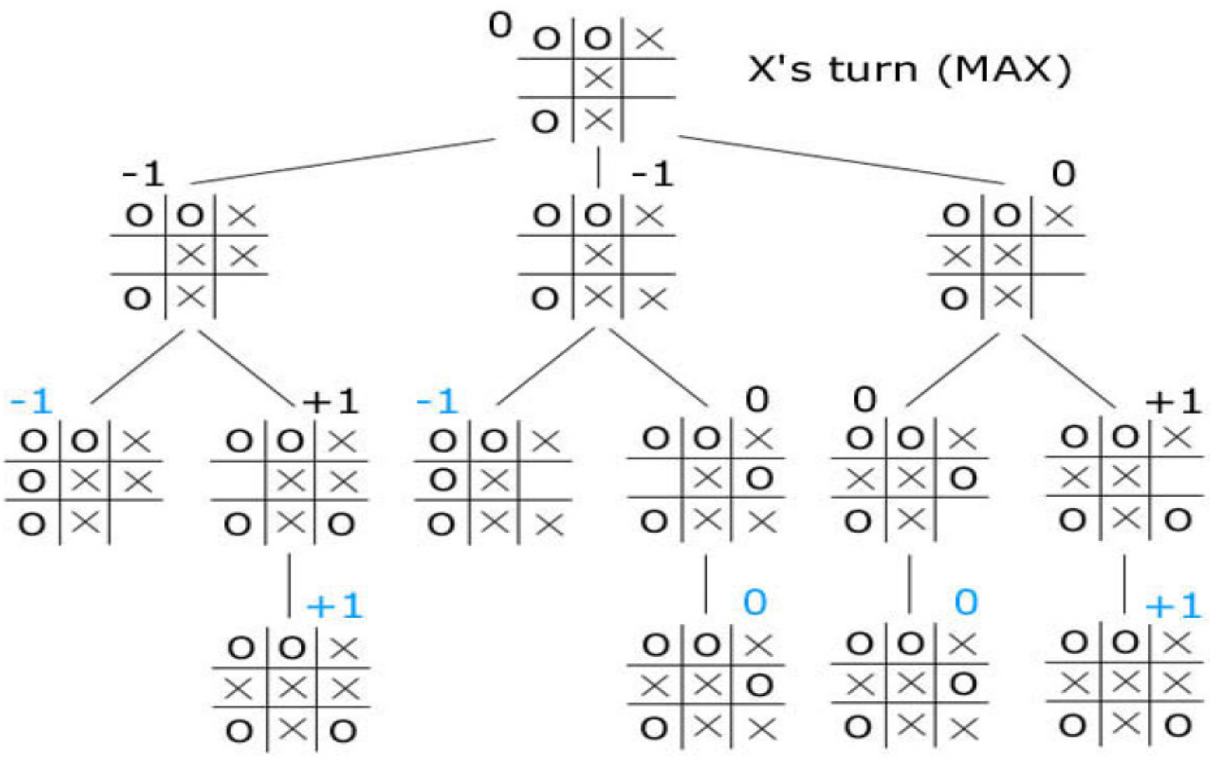
- Two player - we do not deal with coalitions, etc.
- Zero sum - one player's win is the other's loss; there are no cooperative victories

We also consider only perfect information games.

1.12.1 Game Trees

The above category of games can be represented as a tree where the nodes represent the current state of the game and the arcs represent the moves. The game tree consists of all possible moves for the current players starting at the root and all possible moves for the next player as the children of these nodes, and so forth, as far into the future of the game as desired. Each individual move by one player is called a "ply". The leaves of the game tree represent terminal positions as one where the outcome of the game is clear (a win, a loss, a draw, a payoff). Each terminal position has a score. High scores are good for one of the player, called the MAX player. The other player, called MIN player, tries to minimize the score. For example, we may associate 1 with a win, 0 with a draw and -1 with a loss for MAX.

Example: Game of Tic-Tac-Toe



Above is a section of a game tree for tic tac toe. Each node represents a board position, and the children of each node are the legal moves from that position. To score each position, we will give each position which is favorable for player 1 a positive number (the more positive, the more favorable). Similarly, we will give each position which is favorable for player 2 a negative number (the more negative, the more favorable). In our tic tac toe example, player 1 is 'X', player 2 is 'O', and the only three scores we will have are +1 for a win by 'X', -1 for a win by 'O', and 0 for a draw. Note here that the blue scores are the only ones that can be computed by looking at the current position.

1.12.2 Minimax Algorithm

Now that we have a way of representing the game in our program, how do we compute our optimal move? We will assume that the opponent is rational; that is, the opponent can compute moves just as well as we can, and the opponent will always choose the optimal move with the assumption that we, too, will play perfectly. One algorithm for computing the best move is the minimax algorithm:

```
  minimax(player,board) if(game over in current board position) return
winner children = all legal moves for player from this board if(max's turn)
return maximal score of calling minimax on all the children else (min's
turn) return minimal score of calling minimax on all the children
```

If the game is over in the given position, then there is nothing to compute; minimax will simply return the score of the board. Otherwise, minimax will go through each possible child, and (by recursively calling itself) evaluate each possible move. Then, the best possible move will be chosen, where ‘best’ is the move leading to the board with the most positive score for player 1, and the board with the most negative score for player 2.

How long does this algorithm take? For a simple game like tic tac toe, not too long - it is certainly possible to search all possible positions. For a game like Chess or Go however, the running time is prohibitively expensive. In fact, to completely search either of these games, we would first need to develop interstellar travel, as by the time we finish analyzing a move the sun will have gone nova and the earth will no longer exist. Therefore, all real computer games will search, not to the end of the game, but only a few moves ahead. Of course, now the program must determine whether a certain board position is 'good' or 'bad' for a certainly player. This is often done using an evaluation function. This function is the key to a strong computer game. The depth bound search may stop just as things get interesting (e.g. in the middle of a piece exchange in chess. For this reason, the depth bound is usually extended to the end of an exchange to an quiescent state. The search may also tend to postpone bad news until after the depth bound leading to the horizon effect.

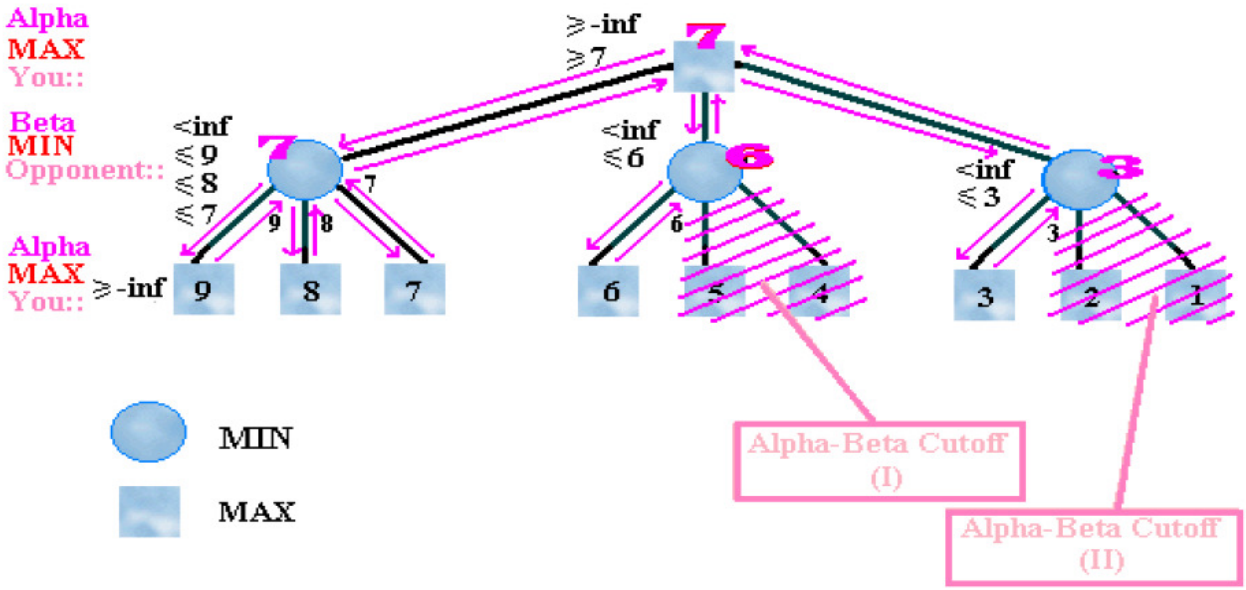
1.12.3 Alpha-Beta Pruning

ALPHA-BETA pruning is a method that reduces the number of nodes explored in Minimax strategy. It reduces the time required for the search and it must be restricted so that no time is to be wasted searching moves that are obviously bad for the current player. The exact implementation of alpha-beta keeps track of the best move for each side as it moves throughout the tree.

We proceed in the same (preorder) way as for the minimax algorithm. For the **MIN** nodes, the score computed starts with **+infinity** and decreases with time. For **MAX** nodes, scores computed starts with **-infinity** and increase with time.

The efficiency of the *Alpha-Beta* procedure depends on the order in which successors of a node are examined. If we were lucky, at a MIN node we would always consider the nodes in order from low to high score and at a MAX node the nodes in order from high to low score. In general it can be shown that in the most favorable circumstances the alpha-beta search opens as many leaves as minimax on a game tree with double its depth.

Here is an example of Alpha-Beta search:



Alpha-Beta algorithm:

The algorithm maintains two values, alpha and beta, which represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of respectively. Initially alpha is negative infinity and beta is positive infinity. As the recursion progresses the "window" becomes smaller.

When beta becomes less than alpha, it means that the current position cannot be the result of best play by both players and hence need not be explored further.

Pseudocode for the alpha-beta algorithm is given below.

```
evaluate (node, alpha, beta) if node is a leaf return the heuristic value
of node if node is a minimizing node for each child of node beta = min
(beta, evaluate (child, alpha, beta)) if beta <= alpha return beta return
beta if node is a maximizing node for each child of node alpha = max
(alpha, evaluate (child, alpha, beta)) if beta <= alpha return alpha
return alpha
```