What is Search?

Search is the systematic examination of **states** to find path from the **start/root state** to the **goal state.** The set of possible states, together with operators defining their connectivity constitute the search space.

The output of a search algorithm is a solution, that is, a path from the initial state to a state that satisfies the goal test.

Search Terminology

- **Problem Space** − It is the environment in which the search takes place. (A set of states and set of operators to change those states)
- **Problem Instance** − It is Initial state + Goal state.
- **Problem Space Graph** − It represents problem state. States are shown by nodes and operators are shown by edges.
- **Depth of a problem** − Length of a shortest path or shortest sequence of operators from Initial State to goal state.
- **Space Complexity** − The maximum number of nodes that are stored in memory.
- **Time Complexity** − The maximum number of nodes that are created.
- **Admissibility** − A property of an algorithm to always find an optimal solution.
- **Branching Factor** − The average number of child nodes in the problem space graph.
- **Depth** − Length of the shortest path from initial state to goal state.

Problem Solving by Search

An important aspect of intelligence is ***goal-based*** problem solving.

The **solution** of many **problems** can be described by finding a **sequence of actions** that lead to a desirable **goal.** Each action changes the ***state*** and the aim is to find the sequence of actions and states that lead from the initial (start) state to a final (goal) state.

- **Initial state**
- **Operator or successor function** - for any state x returns s(x), the set of states reachable from x with one action
- **State space** - all states reachable from initial by any sequence of actions
- **Path** - sequence through state space
- **Path cost** - function that assigns a cost to a path. Cost of a path is the sum of costs of individual actions along the path
- **Goal test** - test to determine if at goal state

.

Problem-solving agents

A Problem solving agent is a **goal-based** agent . It decide what to do by finding sequence of actions that lead to desirable states. The agent can adopt a goal and aim at satisfying it. To illustrate the agent's behavior ,let us take an example where our agent is in the city of Arad,which is in Romania. The agent has to adopt a **goal** of getting to Bucharest.

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 to a goal state.

Problem formulation is the process of deciding what actions and states to consider given a goal.

Example: Route finding problem
Referring to figure 1.19
On holiday in Romania : currently in Arad.
Flight leaves tomorrow from Bucharest
Formulate goal: be in Bucharest

Formulate problem:
states: various cities
actions: drive between cities

Find solution:
sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Problem formulation

A **problem** is defined by four items: **initial state** e.g., "at Arad"
successor function S(x) = set of action-state pairs e.g., S(Arad) = {[Arad -> Zerind;Zerind],….} **goal test**, can be
explicit, e.g., x = at Bucharest"
implicit, e.g., NoDirt(x)
path cost (additive)
e.g., sum of distances, number of actions executed, etc. c(x; a; y) is the step cost, assumed to be >= 0
A **solution** is a sequence of actions leading from the initial state to a goal state.

Figure 1.17 Goal formulation and problem formulation

\

Search

An agent with several immediate options of unknown value can decide what to do by examining different possible sequences of actions that leads to the states of known value,and then choosing the best sequence. The process of looking for sequences actions from the current state to reach the goal state is called **search.**

The **search algorithm** takes a **problem** as **input** and returns a **solution** in the form of **action sequence.** Once a solution is found,the **execution phase** consists of carrying out the recommended action..

Figure 1.18 shows a simple “formulate,search,execute” design for the agent. Once solution has been executed,the agent will formulate a new goal.

```
function SIMPLE-PROBLEM-SOLVING-AGENT( percept) returns an action
inputs : percept, a percept
static: seq, an action sequence, initially empty
        state, some description of the current world state
        goal, a goal, initially null
        problem, a problem formulation
state UPDATE-STATE(state, percept)
if seq is empty then do
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state,
        goal) seq ← SEARCH( problem)
action ← FIRST(seq);
 seq ←REST(seq)
return action
```

Figure 1.18 A Simple problem solving agent. It first formulates a **goal** and a **problem**,searches for a sequence of actions that would solve a problem,and executes the actions one at a time.

- The agent design assumes the Environment is
  - **Static** : The entire process carried out without paying attention to changes that might be occurring in the environment.
  - **Observable :** The initial state is known and the agent’s sensor detects all aspects that are relevant to the choice of action
  - **Discrete :** With respect to the state of the environment and percepts and actions so that alternate courses of action can be taken
  - **Deterministic :** The next state of the environment is completely determined by the current state and the actions executed by the agent. Solutions to the problem are single sequence of actions

An agent carries out its plan with eye closed. This is called an open loop system because ignoring the percepts breaks the loop between the agent and the environment.

Well-defined problems and solutions

A **problem** can be formally defined by **four components**:

- The **initial state** that the agent starts in . The initial state for our agent of example problem is described by *In(Arad)*
- A **Successor Function** returns the possible **actions** available to the agent. Given a state x,SUCCESSOR-FN(x) returns a set of {action,successor} ordered pairs where 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.

For example,from the state In(Arad),the successor function for the Romania problem would return
{ [Go(Sibiu),In(Sibiu)],[Go(Timisoara),In(Timisoara)],[Go(Zerind),In(Zerind)] }

- **State Space** : The set of all states reachable from the initial state. The state space forms a graph in which the nodes are states and the arcs between nodes are actions.
- A **path** in the state space is a sequence of states connected by a sequence of actions.
- Thr **goal test** determines whether the given state is a goal state.

- A **path cost** function assigns numeric cost to each action. For the Romania problem the cost of path might be its length in kilometers.
- The **step cost** of taking action a to go from state x to state y is denoted by c(x,a,y). The step cost for Romania are shown in figure 1.18. It is assumed that the step costs are non negative.
- A **solution** to the problem is a path from the initial state to a goal state.
- An **optimal solution** has the lowest path cost among all solutions.

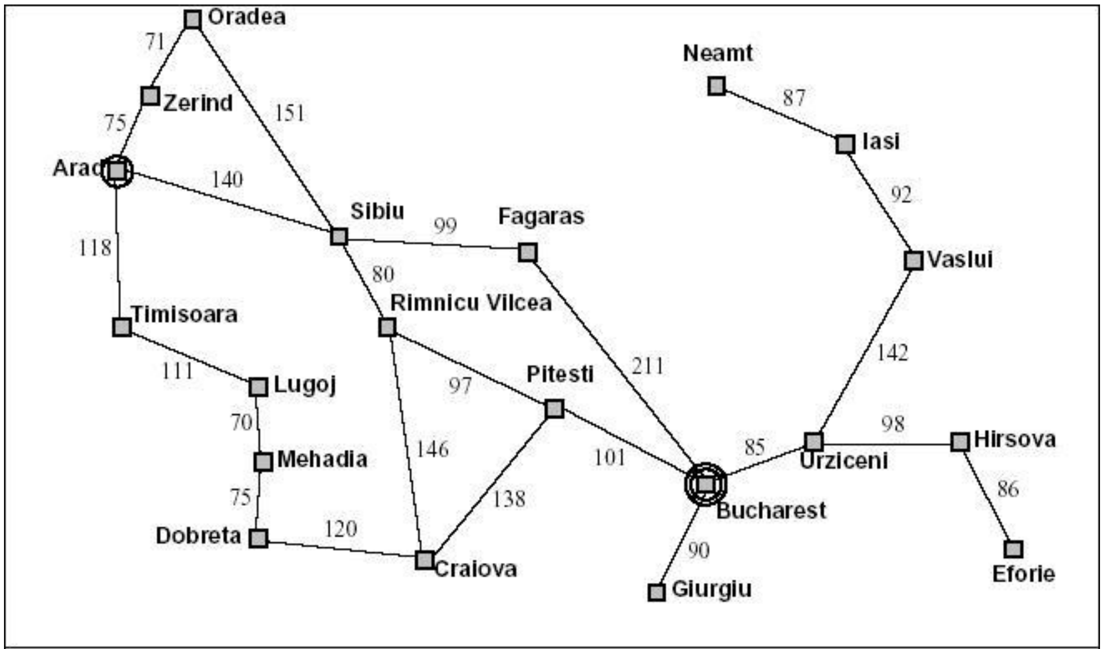


Figure 1.19 A simplified Road Map of part of Romania

EXAMPLE PROBLEMS

The problem solving approach has been applied to a vast array of task environments. Some best known problems are summarized below. They are distinguished as toy or real-world problems

A **toy problem** is intended to illustrate various problem solving methods. It can be easily used by different researchers to compare the performance of algorithms.
A **real world problem** is one whose solutions people actually care about.

Vacuum World Example

- **States**: The agent is in one of two locations.,each of which might or might not contain dirt. Thus there are 2 x 2^2 = 8 possible world states.
- **Initial state**: Any state can be designated as initial state.
- **Successor function** : This generates the legal states that results from trying the three actions (left, right, suck). The complete state space is shown in figure 2.3
- **Goal Test** : This tests whether all the squares are clean.
- **Path test** : Each step costs one ,so that the the path cost is the number of steps in the path.

Vacuum World State Space

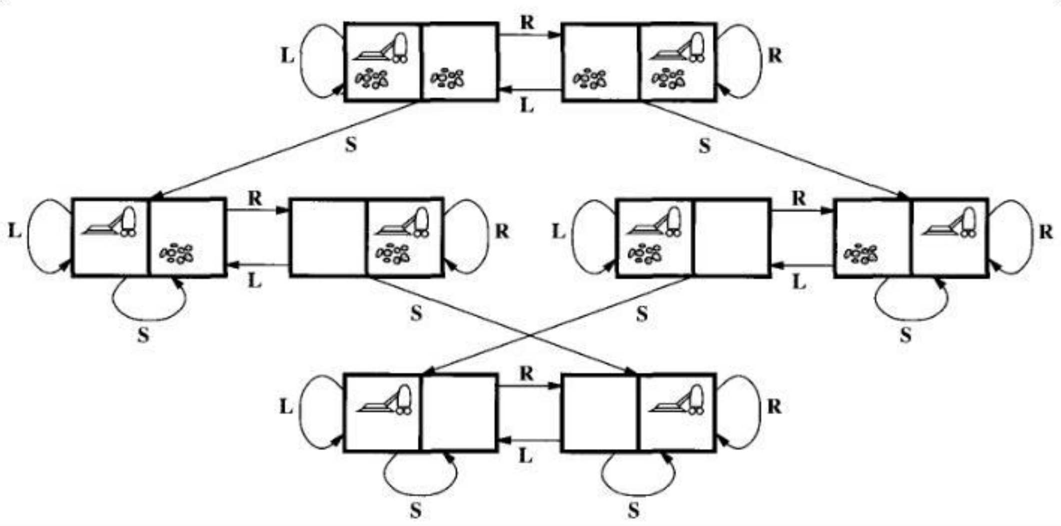


Figure 1.20 The state space for the vacuum world.
Arcs denote actions: L = Left,R = Right,S = Suck

The 8-puzzle

An 8-puzzle consists of a 3x3 board with eight numbered tiles and a blank space. A tile adjacent to the balank space can slide into the space. The object is to reach the goal state ,as shown in figure 2.4

Example: The 8-puzzle

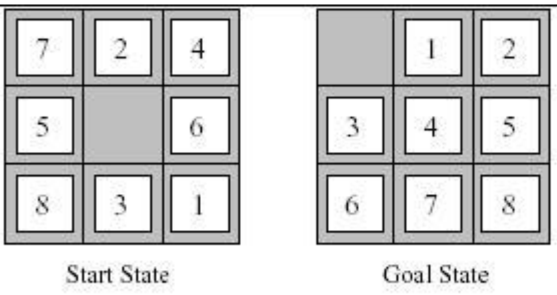


Figure 1.21 A typical instance of 8-puzzle.

The problem formulation is as follows :

- **States** : A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
- **Initial state** : Any state can be designated as the initial state. It can be noted that any given goal can be reached from exactly half of the possible initial states.
- **Successor function** : This generates the legal states that result from trying the four actions(blank moves Left,Right,Up or down).
- **Goal Test** : This checks whether the state matches the goal configuration shown in figure 2.4.(Other goal configurations are possible)
- **Path cost** : Each step costs 1,so the path cost is the number of steps in the path. o

The 8-puzzle belongs to the family **of sliding-block puzzles**,which are often used as test problems for new search algorithms in AI. This general class is known as NP-complete. The **8-puzzle** has 9!/2 = 181,440 reachable states and is easily solved.

The **15 puzzle** (4 x 4 board) has around 1.3 trillion states,an the random instances can be solved optimally in few milli seconds by the best search algorithms.
The **24-puzzle** (on a 5 x 5 board) has around 10^{25} states ,and random instances are still quite difficult to solve optimally with current machines and algorithms.

8-queens problem

The goal of 8-queens problem is to place 8 queens on the chessboard such that no queen attacks any other.(A queen attacks any piece in the same row,column or diagonal). Figure 2.5 shows an attempted solution that fails: the queen in the right most column is attacked by the queen at the top left.

An **Incremental formulation** involves operators that augments the state description,starting with an empty state.for 8-queens problem,this means each action adds a queen to the state.
A **complete-state formulation** starts with all 8 queens on the board and move them around.
In either case the path cost is of no interest because only the final state counts.

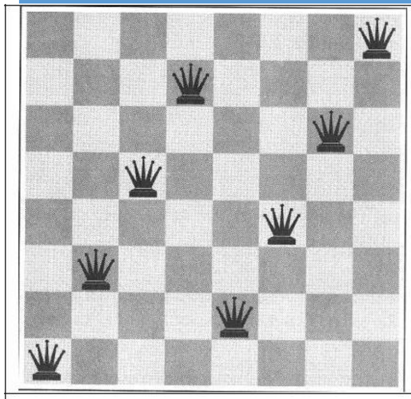


Figure 1.22 8-queens problem

The first incremental formulation one might try is the following :

o **States** : Any arrangement of 0 to 8 queens on board is a state.
o **Initial state** : No queen on the board.
o **Successor function** : Add a queen to any empty square.
o **Goal Test** : 8 queens are on the board,none attacked.

In this formulation,we have 64.63…57 = 3 x 10^{14} possible sequences to investigate.
A better formulation would prohibit placing a queen in any square that is already attacked.
:

- o **States** : Arrangements of n queens (0 <= n < = 8) ,one per column in the left most columns ,with no queen attacking another are states.
- o **Successor function** : Add a queen to any square in the left most empty column such that it is not attacked by any other queen.

This formulation reduces the 8-queen state space from 3 x 10^{14} to just 2057,and solutions are easy to find.
For the 100 queens the initial formulation has roughly 10^{400} states whereas the improved formulation has about 10^{52} states. This is a huge reduction,but the improved state space is still too big for the algorithms to handle.

REAL-WORLD PROBLEMS

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 air line travel planning systems.

AIRLINE TRAVEL PROBLEM

The **airline travel problem** is specifies as follows **:**

o **States :** Each is represented by a location(e.g.,an airport) and the current time. o **Initial state :** This is specified by the problem.

- o **Successor function :** This returns the states resulting from taking any scheduled flight(further specified by seat class and location),leaving later than the current time plus the within-airport transit time,from the current airport to another.

o **Goal Test :** Are we at the destination by some prespecified time?

- **Path cost :** This depends upon the monetary cost,waiting time,flight time,customs and immigration procedures,seat quality,time of dat,type of air plane,frequent-flyer mileage awards, and so on.

TOURING PROBLEMS

Touring problems are closely related to route-finding problems,but with an important difference. Consider for example,the problem,"Visit every city at least once" as shown in Romania map.

As with route-finding the actions correspond to trips between adjacent cities. The state space, however,is quite different.
The initial state would be "In Bucharest; visited{Bucharest}".
A typical intermediate state would be "In Vaslui;visited {Bucharest,Urziceni,Vaslui}".
The goal test would check whether the agent is in Bucharest and all 20 cities have been visited.

THE TRAVELLING SALESPERSON PROBLEM(TSP)

Is a touring problem in which each city must be visited exactly once. The aim is to find the shortest tour.The problem is known to be **NP-hard**. Enormous efforts have been expended to improve the capabilities of TSP algorithms. These algorithms are also used in tasks such as planning movements of **automatic circuit-board drills** and of **stocking machines** on shop floors.

VLSI layout

A **VLSI layout** problem requires positioning millions of components and connections on a chip to minimize area ,minimize circuit delays,minimize stray capacitances,and maximize manufacturing yield. The layout problem is split into two parts : **cell layout** and **channel routing**.

ROBOT navigation

ROBOT navigation is a generalization of the route-finding 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. For a circular Robot moving on a flat surface,the space is essentially two-dimensional. When the robot has arms and legs or wheels that also must be controlled,the search space becomes multi-dimensional. Advanced techniques are required to make the search space finite.

AUTOMATIC ASSEMBLY SEQUENCING

The example includes assembly of intricate objects such as electric motors. The aim in assembly problems is to find the order in which to assemble the parts of some objects. If the wrong order is choosen,there will be no way to add some part later without undoing somework already done. Another important assembly problem is protein design,in which the goal is to find a sequence of Amino acids that will be fold into a three-dimensional protein with the right properties to cure some disease.

INTERNET SEARCHING

In recent years there has been increased demand for software robots that perform Internet searching.,looking for answers to questions,for related information,or for shopping deals. The searching techniques consider internet as a graph of nodes(pages) connected by links.

Example: Water Jug Problem

A Water Jug Problem: You are given two jugs, a 4-gallon one and a 3-gallon one, a pump which has unlimited water which you can use to ll the jug, and the ground on which water may be poured. Neither jug has any measuring markings on it. How can you get exactly 2 gallons of water in the 4-gallon jug?

State Representation and Initial State { we will represent a state of the problem as a tuple (x, y) where x represents the amount of water in the 4-gallon jug and y represents the amount of water in the 3-gallon jug. Note 0 x 4, and 0 y 3. Our initial state: (0,0)

Goal Predicate { state = (2,y) where 0 y 3.

Operators { we must de ne a set of operators that will take us from one state to another:

1. Fill 4-gal jug	(x,y)	! (4,y)
	x < 4	
2. Fill 3-gal jug	(x,y)	! (x,3)
	y < 3	
3. Empty 4-gal jug on ground	(x,y)	! (0,y)
	x > 0	
4. Empty 3-gal jug on ground	(x,y)	! (x,0)
	y > 0	
5. Pour water from 3-gal jug	(x,y)	! (4, y - (4 - x))
to ll 4-gal jug	0 < x+y 4 and y > 0	
6. Pour water from 4-gal jug	(x,y)	! (x - (3-y), 3)
to ll 3-gal-jug	0 < x+y 3 and x > 0	
7. Pour all of water from 3-gal jug	(x,y)	! (x+y, 0)
into 4-gal jug	0 < x+y 4 and y 0	
8. Pour all of water from 4-gal jug	(x,y)	! (0, x+y)
into 3-gal jug	0 < x+y 3 and x 0	

Through Graph Search, the following solution is found :

Gals in 4-gal jug	Gals in 3-gal jug	Rule Applied
0	0	
		1. Fill 4
4	0	
		6. Pour 4 into 3 to ll
1	3	
		4. Empty 3
1	0	
		8. Pour all of 4 into 3
0	1	
		1. Fill 4
4	1	
		6. Pour into 3
2	3	

Example: Tic-Tac-Toe:

Tic-tac-toe is not a very challenging game for human beings. If you're an enthusiast, you've probably moved from the basic game to some variant like three-dimensional tic-tac-toe on a larger grid.

If you sit down right now to play ordinary three-by-three tic-tac-toe with a friend, what will probably happen is that every game will come out a tie. Both you and your friend can probably play perfectly, never making a mistake that would allow your opponent to win.

But can you *describe* how you know where to move each turn? Most of the time, you probably aren't even aware of alternative possibilities; you just look at the board and instantly know where you want to move. That kind of instant knowledge is great for human beings, because it makes you a fast player. But it isn't much help in writing a computer program. For that, you have to know very explicitly what your strategy is.

By the way, although the example of tic-tac-toe strategy is a relatively trivial one, this issue of instant knowledge versus explicit rules is a hot one in modern psychology. Some cognitive scientists, who think that human intelligence works through mechanisms similar to computer programs, maintain that when you know how to do something without knowing *how* you know, you have an explicit set of rules deep down inside. It's just that the rules have become a habit, so you don't think about them deliberately.
They're "compiled," in the jargon of cognitive psychology. On the other hand, some people think that your implicit how-to knowledge is very different from the sort of lists of rules that can be captured in a computer program. They think that human thought is profoundly different from the way computers work, and that a

computer cannot be programmed to simulate the full power of human problem-solving. These people would say, for example, that when you look at a tic-tac-toe board you immediately grasp the strategic situation as a whole, and your eye is drawn to the best move without any need to examine alternatives according to a set of rules. (You might like to try to be aware of your own mental processes as you play a game of tic-tac-toe, to try to decide which of these points of view more closely resembles your own experience—but on the other hand, the psychological validity of such introspective evidence is *another* hotly contested issue in psychology!)

☞ Before you read further, try to write down a set of strategy rules that, if followed consistently, will never lose a game. Play a few games using your rules. Make sure they work even if the other player does something bizarre.

I'm going to number the squares in the tic-tac-toe board this way:

```
1 2 3
4 5 6
7 8 9
```

Squares 1, 3, 7, and 9 are *corner squares*. I'll call 2, 4, 6, and 8 *edge squares*. And of course number 5 is the *center square*. I'll use the word *position* to mean a specific partly-filled-in board with X and O in certain squares, and other squares empty.

One way you might meet my challenge of describing your strategy explicitly is to list all the possible sequences of moves up to a certain point in the game, then say what move you'd make next in each situation. How big would the list have to be? There are nine possibilities for the first move. For each first move, there are eight possibilities for the second move. If you continue this line of reasoning, you'll see that there are nine factorial, or 362880, possible sequences of moves. Your computer may not have enough memory to list them all, and you certainly don't have enough patience!

Fortunately, not all these sequences are interesting. Suppose you are describing the rules a computer should use against a human player, and suppose the human being moves first. Then there are, indeed, nine possible first moves. But for each of these, there is only *one* possible computer move! After all, we're programming the computer. We get to decide which move it will choose. Then there are seven possible responses by the opponent, and so on. The number of sequences when the human being plays first is 9 times 7 times 5 times 3, or 945. If the computer plays first, it will presumably always make the single best choice.

Then there are eight possible responses, and so on. In this case the number of possible game sequences is 8 times 6 times 4 times 2, or 384. Altogether we have 1329 cases to worry about, which is much better than 300,000 but still not an enjoyable way to write a computer program.

In fact, though, this number is still too big. Not all games go for a full nine moves before someone wins. Also, many moves force the opponent to a single possible response, even though there are other vacant squares on the board. Another reduction can be achieved by taking advantage of *symmetry*. For example, if X starts in square 5, any game sequence in which O responds in square 1 is equivalent to a sequence in which O responds in square 3, with the board rotated 90 degrees. In fact there are only two truly different responses to a center-square opening: any corner square, or any edge square.

With all of these factors reducing the number of distinct positions, it would probably be possible to list all of them and write a strategy program that way. I'm not sure, though, because I didn't want to use that technique. I was looking for rules expressed in more general terms, like "all else being equal, pick a corner rather than an edge."

Why should I prefer a corner? Each corner square is part of three winning combinations. For example, square 1 is part of **123**, **147**, and **159**. (By expressing these winning combinations as three-digit numbers, I've jumped ahead a bit in the story with a preview of how the program I wrote represents this information.) An edge square, on the other hand, is only part of two winning combinations. For example, square 2 is part of **123** and **258**. Taking a corner square makes three winning combinations available to me and unavailable to my opponent.

Since I've brought up the subject of winning combinations, how many of *them* are there? Not very many: three horizontal, three vertical, and two diagonal. Eight altogether. That *is* a reasonable amount of information to include in a program, and in fact there is a list of the eight winning combinations in this project.

You might, at this point, enjoy playing a few games with the program, to see if you can figure out the rules it uses in its strategy. If you accepted my earlier challenge to write down your own set of strategy rules, you can compare mine to yours. Are they the same? If not, are they equally good?

The top-level procedure in this project is called **ttt**. It takes no inputs. When you invoke this procedure, it will ask you if you'd like to play first (X) or second (O). Then you enter moves by typing a digit 1–9 for the square you select. The program draws the game board on the Logo graphics screen.

I'm about to start explaining my strategy rules, so stop reading if you want to work out your own and haven't done it yet.

Strategy

The highest-priority and the lowest-priority rules seemed obvious to me right away. The highest-priority are these:

1. If I can win on this move, do it.
2. If the other player can win on the next move, block that winning square.

Here are the lowest-priority rules, used only if there is nothing suggested more strongly by the board position:

n − 2. Take the center square if it's free.

n − 1. Take a corner square if one is free.

n. Take whatever is available.

The highest priority rules are the ones dealing with the most urgent situations: either I or my opponent can win on the next move. The lowest priority ones deal with the least urgent situations, in which there is nothing special about the moves already made to guide me.

What was harder was to find the rules in between. I knew that the goal of my own tic-tac-toe strategy was to set up a *fork*, a board position in which I have two winning moves, so my opponent can only block one of them. Here is an example:

```
x o

  x

x   o
```

X can win by playing in square 3 or square 4. It's O's turn, but poor O can only block one of those squares at a time. Whichever O picks, X will then win by picking the other one.

Given this concept of forking, I decided to use it as the next highest priority rule:

3. If I can make a move that will set up a fork for myself, do it.

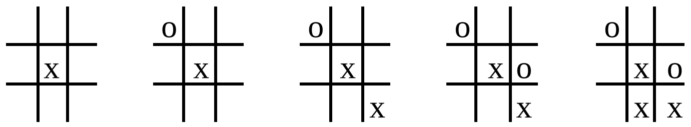
That was the end of the easy part. My first attempt at writing the program used only these six rules. Unfortunately, it lost in many different situations. I needed to add something, but I had trouble finding a good rule to add.

My first idea was that rule 4 should be the defensive equivalent of rule 3, just as rule 2 is the defensive equivalent of rule 1:

4a. If, on the next move, my opponent can set up a fork, block that possibility by moving into the square that is common to his two winning combinations.

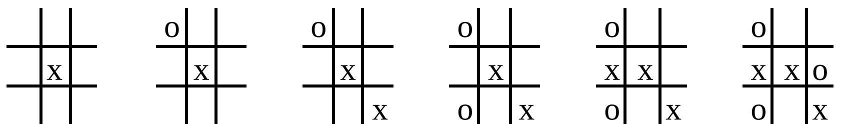
In other words, apply the same search technique to the opponent's position that I applied to my own.

This strategy works well in many cases, but not all. For example, here is a sequence of moves under this strategy, with the human player moving first:



In the fourth grid, the computer (playing O) has discovered that X can set up a fork by moving in square 6, between the winning combinations **456** and **369**. The computer moves to block this fork. Unfortunately, X can also set up a fork by moving in squares 3, 7, or 8. The computer's move in square 6 has blocked one combination of the square-3 fork, but X can still set up the other two. In the fifth grid, X has moved in square 8. This sets up the winning combinations **258** and **789**. The computer can only block one of these, and X will win on the next move.

Since X has so many forks available, hopeless before O moved in square 6? No. does this mean that the game was already something O could have done: Here is



In this sequence, the computer's second move is in square 7. This move also blocks a fork, but it wasn't chosen for that reason. Instead, it was chosen *to force X's next move*. In the fifth grid, X has had to move in square 4, to prevent an immediate win by O. The advantage of this situation for O is that square 4 was *not* one of the ones with which X could set up a fork. O's next move, in the sixth grid, is also forced. But by then the board is too crowded for either player to force a win; the game ends in a tie, as usual.

This analysis suggests a different choice for an intermediate-level strategy rule, taking the offensive:

4b. If I can make a move that will set up a winning combination for myself, do it.

Compared to my earlier try, this rule has the benefit of simplicity. It's much easier for the program to look for a single winning combination than for a fork, which is two such combinations with a common square.

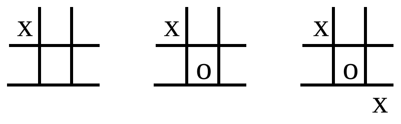
Unfortunately, this simple rule isn't quite good enough. In the example just above, the computer found the winning combination **147** in which it already had square 1, and the other two were free. But why should it choose to move in square 7 rather than square 4? If the program did choose square 4, then X's move would still be forced, into square 7.
We would then have forced X into creating a fork, which would defeat the program on the next move.

It seems that there is no choice but to combine the ideas from rules 4a and 4b:

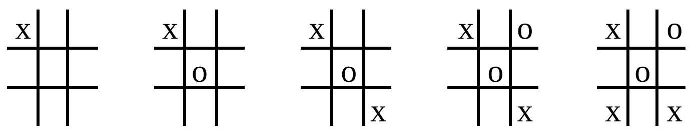
4. If I can make a move that will set up a winning combination for myself, do it. But ensure that this move does not force the opponent into establishing a fork.

What this means is that we are looking for a winning combination in which the computer already owns one square and the other two are empty. Having found such a combination, we can move in either of its empty squares. Whichever we choose, the opponent will be forced to choose the other one on the next move. If one of the two empty squares would create a fork for the opponent, then the computer must choose that square and leave the other for the opponent.

What if *both* of the empty squares in the combination we find would make forks for the opponent? In that case, we've chosen a bad winning combination. It turns out that there is only one situation in which this can happen:



Again, the computer is playing O. After the third grid, it is looking for a possible winning combination for itself. There are three possibilities: **258**, **357**, and **456**. So far we have not given the computer any reason to prefer one over another. But here is what happens if the program happens to choose **357**:



By this choice, the computer has forced its opponent into a fork that will win the game for the opponent. If the computer chooses either of the other two possible winning combinations, the game ends in a tie. (All moves after this choice turn out to be forced.)

This particular game sequence was very troublesome for me because it goes against most of the rules I had chosen earlier. For one thing, the correct choice for the program is any edge square, while the corner squares must be avoided. This is the opposite of the usual priority.

Another point is that this situation contradicts rule 4a (prevent forks for the other player) even more sharply than the example we considered earlier. In that example, rule 4a wasn't enough guidance to ensure a correct choice, but the correct choice was at least *consistent* with the rule. That is, just blocking a fork isn't enough, but threatening a win and *also* blocking a fork is better than just threatening a win alone. This is the meaning of rule 4. But in this new situation, the corner square (the move we have to avoid) *does* block a fork, while the edge square (the correct move) *doesn't* block a fork!

8 Puzzle

Write a program to solve the 8-puzzle problem (and its natural generalizations) using the A* search algorithm.

The problem. The 8-puzzle problem is a puzzle invented and popularized by Noyes Palmer Chapman in the 1870s. It is played on a 3-by-3 grid with 8 square blocks labeled 1 through 8 and a blank square. Your goal is to rearrange the blocks so that they are in order. You are permitted to slide blocks horizontally or vertically into the blank square. The following shows a sequence of legal moves from an initial board position (left) to the goal position (right).

```
   1  3       1   3       1  2  3      1  2  3      1  2  3
 4  2  5  =>  4  2  5  =>  4     5  =>  4  5     =>  4  5  6
 7  8  6      7  8  6      7  8  6      7  8  6      7  8

 initial                                        goal
```

SEARCHING FOR SOLUTIONS

SEARCH TREE

Having formulated some **problems**,we now need to **solve** them. This is done by a **search** through the **state space**. A **search tree** is generated by the **initial state** and the **successor function** that together define the **state space**. In general,we may have a *search graph* rather than a *search tree*,when the same state can be reached from multiple paths.

UNINFORMED SEARCH STRATGES

Uninformed Search Strategies have no additional information about states beyond that provided in the **problem definition**.
Strategies that know whether one non goal state is "more promising" than another are called **Informed search or heuristic search** strategies.

There are five uninformed search strategies as given below.

- o Breadth-first search
- o Uniform-cost search
- o Depth-first search
- o Depth-limited search
- o Iterative deepening search

Breadth-first search

Breadth-first search is a simple strategy in which the root node is expanded first,then all successors of the root node are expanded next,then their successors,and so on. In general,all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.

Breath-first-search is implemented by calling TREE-SEARCH with an empty fringe that is a first-in-first-out(FIFO) queue,assuring that the nodes that are visited first will be expanded first. In otherwards,calling TREE-SEARCH(problem,FIFO-QUEUE()) results in breadth-first-search. The FIFO queue puts all newly generated successors at the end of the queue,which means that Shallow nodes are expanded before deeper nodes.

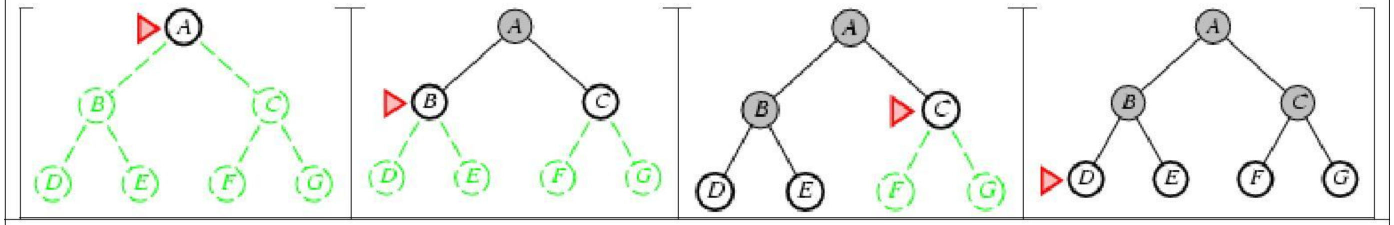
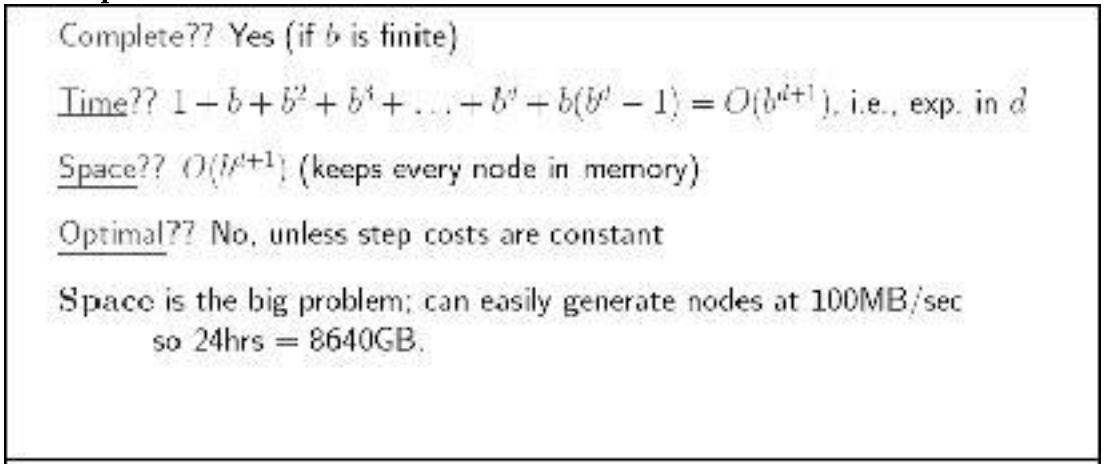


Figure 1.27 Breadth-first search on a simple binary tree. At each stage ,the node to be expanded next is indicated by a marker.

Properties of breadth-first-search



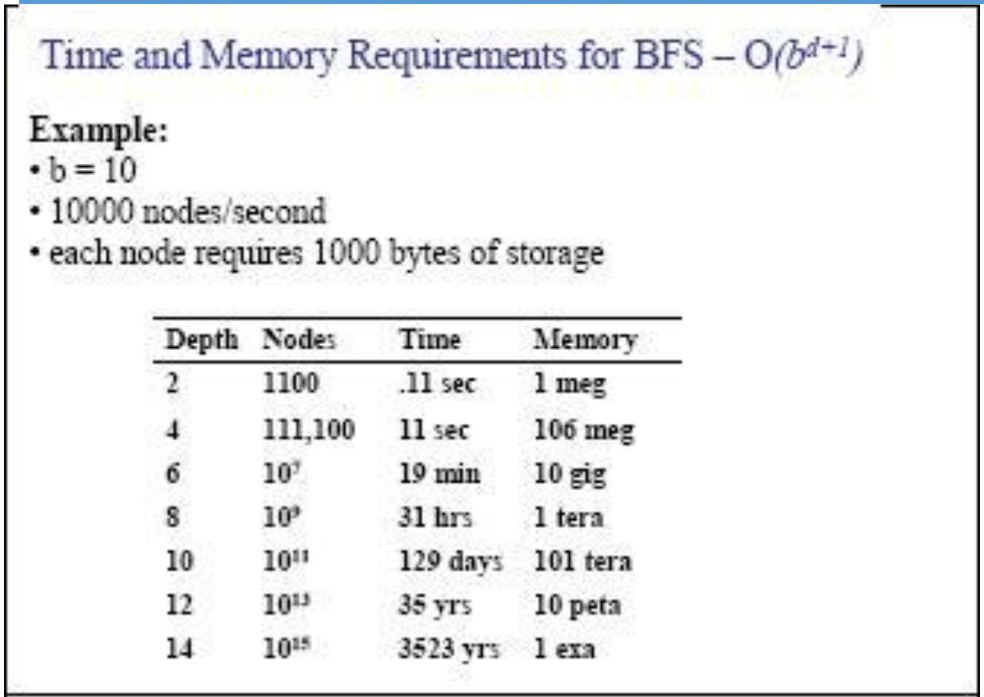
Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d

Space?? $O(b^{d+1})$ (keeps every node in memory)

Optimal?? No, unless step costs are constant

Space is the big problem; can easily generate nodes at 100MB/sec
so 24hrs = 8640GB.



Time and Memory Requirements for BFS – $O(b^{d+1})$

Example:

- b = 10
- 10000 nodes/second
- each node requires 1000 bytes of storage

Depth	Nodes	Time	Memory
2	1100	.11 sec	1 meg
4	111,100	11 sec	106 meg
6	10^7	19 min	10 gig
8	10^9	31 hrs	1 tera
10	10^{11}	129 days	101 tera
12	10^{13}	35 yrs	10 peta
14	10^{15}	3523 yrs	1 exa

Figure 1.29 Time and memory requirements for breadth-first-search. The numbers shown assume branch factor of b = 10 ; 10,000 nodes/second; 1000 bytes/node

Time complexity for BFS

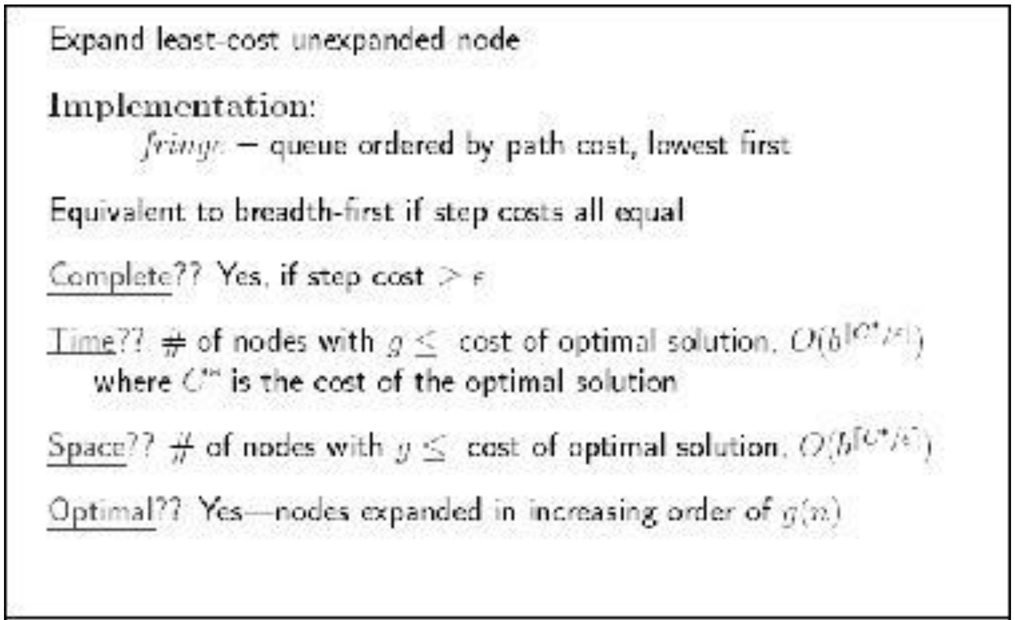
Assume every state has b successors. The root of the search tree generates b nodes at the first level,each of which generates b more nodes,for a total of b^2 at the second level. Each of these generates b more nodes,yielding b^3 nodes at the third level,and so on. Now suppose,that the solution is at depth d. In the worst case,we would expand all but the last node at level d,generating $b^{d+1} - b$ nodes at level d+1.

Then the total number of nodes generated is
$b + b^2 + b^3 + \ldots + b^d + (b^{d+1} + b) = O(b^{d+1}).$

Every node that is generated must remain in memory,because it is either part of the fringe or is an ancestor of a fringe node. The space compleity is,therefore ,the same as the time complexity

UNIFORM-COST SEARCH

Instead of expanding the shallowest node,**uniform-cost search** expands the node n with the lowest path cost. uniform-cost search does not care about the number of steps a path has,but only about their total cost.



Expand least-cost unexpanded node

Implementation:
fringe = queue ordered by path cost, lowest first

Equivalent to breadth-first if step costs all equal

Complete?? Yes, if step cost $\geq \epsilon$

Time?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$
where C^* is the cost of the optimal solution

Space?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$

Optimal?? Yes—nodes expanded in increasing order of $g(n)$

Figure 1.30 Properties of Uniform-cost-search

DEPTH-FIRST-SEARCH

Depth-first-search always expands the deepest node in the current fringe of the search tree. The progress of the search is illustrated in figure 1.31. The search proceeds immediately to the deepest level of the search tree,where the nodes have no successors. As those nodes are expanded,they are dropped from the fringe,so then the search “backs up” to the next shallowest node that still has unexplored successors.

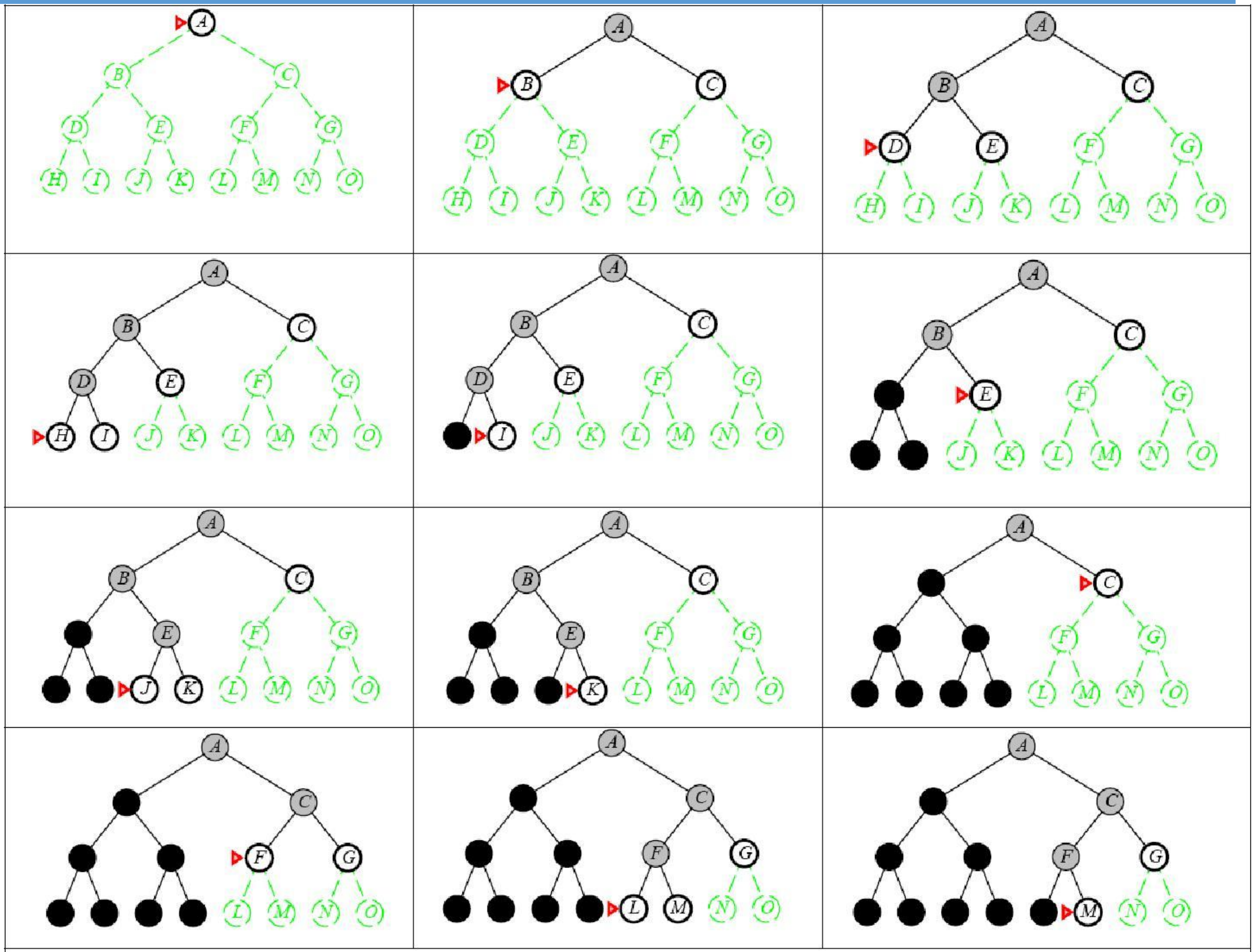


Figure 1.31 Depth-first-search on a binary tree. Nodes that have been expanded and have no descendants in the fringe can be removed from the memory;these are shown in black. Nodes at depth 3 are assumed to have no successors and M is the only goal node.

This strategy can be implemented by TREE-SEARCH with a last-in-first-out (LIFO) queue,also known as a stack.

Depth-first-search has very modest memory requirements.It needs to store only a single path from the root to a leaf node,along with the remaining unexpanded sibling nodes for each node on the path. Once the node has been expanded,it can be removed from the memory,as soon as its descendants have been fully explored(Refer Figure 2.12).

For a state space with a branching factor b and maximum depth m,depth-first-search requires storage of only bm + 1 nodes.

Using the same assumptions as Figure 2.11,and assuming that nodes at the same depth as the goal node have no successors,we find the depth-first-search would require 118 kilobytes instead of 10 petabytes,a factor of 10 billion times less space.

Drawback of Depth-first-search

The drawback of depth-first-search is that it can make a wrong choice and get stuck going down very long(or even infinite) path when a different choice would lead to solution near the root of the search tree. For example ,depth-first-search will explore the entire left subtree even if node C is a goal node.

BACKTRACKING SEARCH

A variant of depth-first search called backtracking search uses less memory and only one successor is generated at a time rather than all successors.; Only O(m) memory is needed rather than O(bm)

DEPTH-LIMITED-SEARCH

The problem of unbounded trees can be alleviated by supplying depth-first-search with a pre-determined depth limit l.That is,nodes at depth l are treated as if they have no successors. This approach is called **depth-limited-search**. The depth limit soves the infinite path problem. Depth limited search will be nonoptimal if we choose l > d. Its time complexity is $O(b^l)$ and its space compleiy is O(bl). Depth-first-search can be viewed as a special case of depth-limited search with l = oo

Sometimes,depth limits can be based on knowledge of the problem. For,example,on the map of Romania there are 20 cities. Therefore,we know that if there is a solution.,it must be of length 19 at the longest,So l = 10 is a possible choice. However,it oocan be shown that any city can be reached from any other city in at most 9 steps. This number known as the **diameter** of the state space,gives us a better depth limit.

Depth-limited-search can be implemented as a simple modification to the general tree-search algorithm or to the recursive depth-first-search algorithm. The pseudocode for recursive depth-limited-search is shown in Figure 1.32.

It can be noted that the above algorithm can terminate with two kinds of failure : the standard *failure* value indicates no solution; the *cutoff* value indicates no solution within the depth limit. Depth-limited search = depth-first search with depth limit l,
returns cut off if any path is cut off by depth limit

```
function Depth-Limited-Search( problem, limit) returns a solution/fail/cutoff
return Recursive-DLS(Make-Node(Initial-State[problem]), problem, limit)
function Recursive-DLS(node, problem, limit) returns solution/fail/cutoff
cutoff-occurred? ⟵ false
if Goal-Test(problem,State[node]) then return Solution(node)
else if Depth[node] = limit then return cutoff
else for each successor in Expand(node, problem) do
result ⟵ Recursive-DLS(successor, problem, limit)
if result = cutoff then cutoff_occurred? ⟵ true else
if result not = failure then return result
if cutoff_occurred? then return cutoff else return failure
```

Figure 1.32 Recursive implementation of Depth-limited-search:

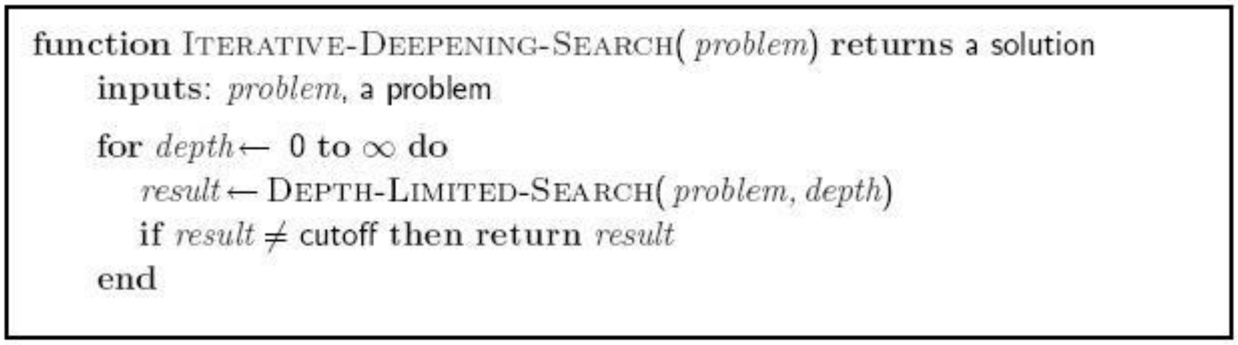
ITERATIVE DEEPENING DEPTH-FIRST SEARCH

Iterative deepening search (or iterative-deepening-depth-first-search) is a general strategy often used in combination with depth-first-search,that finds the better depth limit. It does this by gradually increasing the limit – first 0,then 1,then 2, and so on – until a goal is found. This will occur when the depth limit reaches d,the depth of the shallowest goal node. The algorithm is shown in Figure 2.14.

Iterative deepening combines the benefits of depth-first and breadth-first-search
Like depth-first-search,its memory requirements are modest;O(bd) to be
precise.

Like Breadth-first-search,it is complete when the branching factor is finite and optimal when the path cost is a non decreasing function of the depth of the node.

Figure 2.15 shows the four iterations of ITERATIVE-DEEPENING_SEARCH on a binary search tree,where the solution is found on the fourth iteration.



```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution
    inputs: problem, a problem

    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
    end
```

Figure 1.33 The **iterative deepening search algorithm** ,which repeatedly applies depth-limited-search with increasing limits. It terminates when a solution is found or if the depth limited search resturns *failure*,meaning that no solution exists.

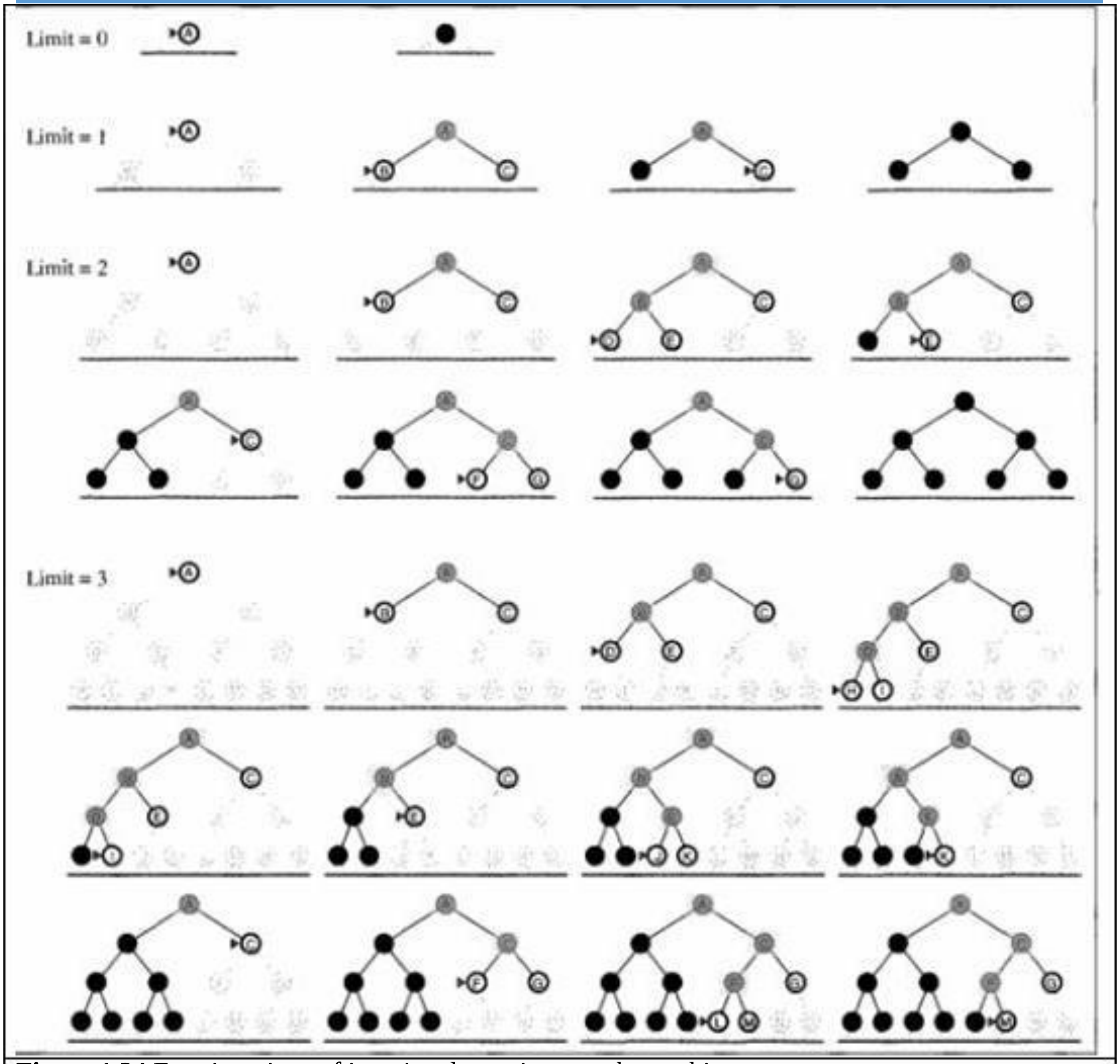


Figure 1.34 Four iterations of iterative deepening search on a binary tree

Iterative search is not as wasteful as it might seem

Iterative deepening search

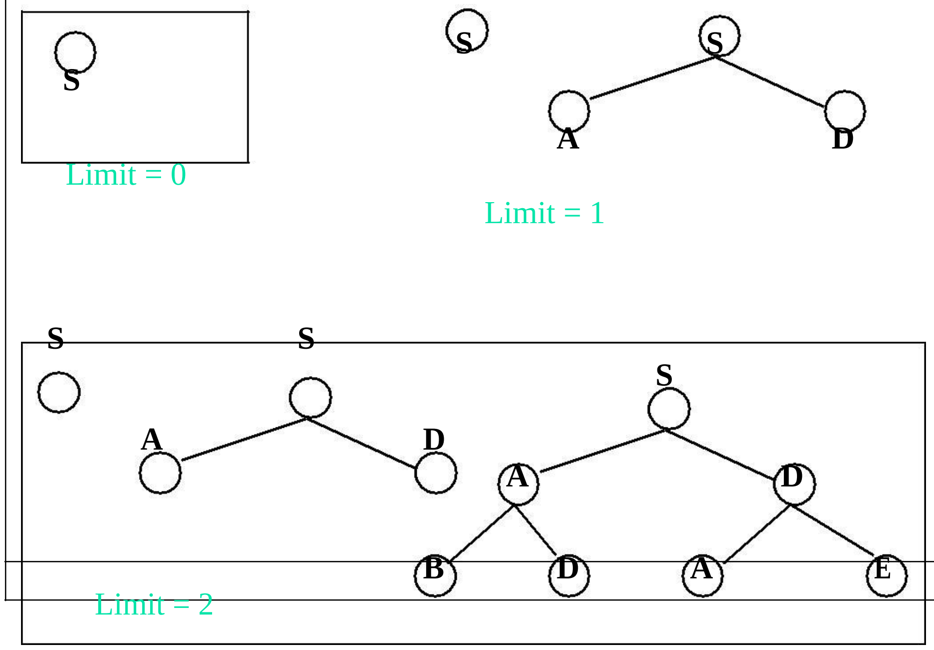
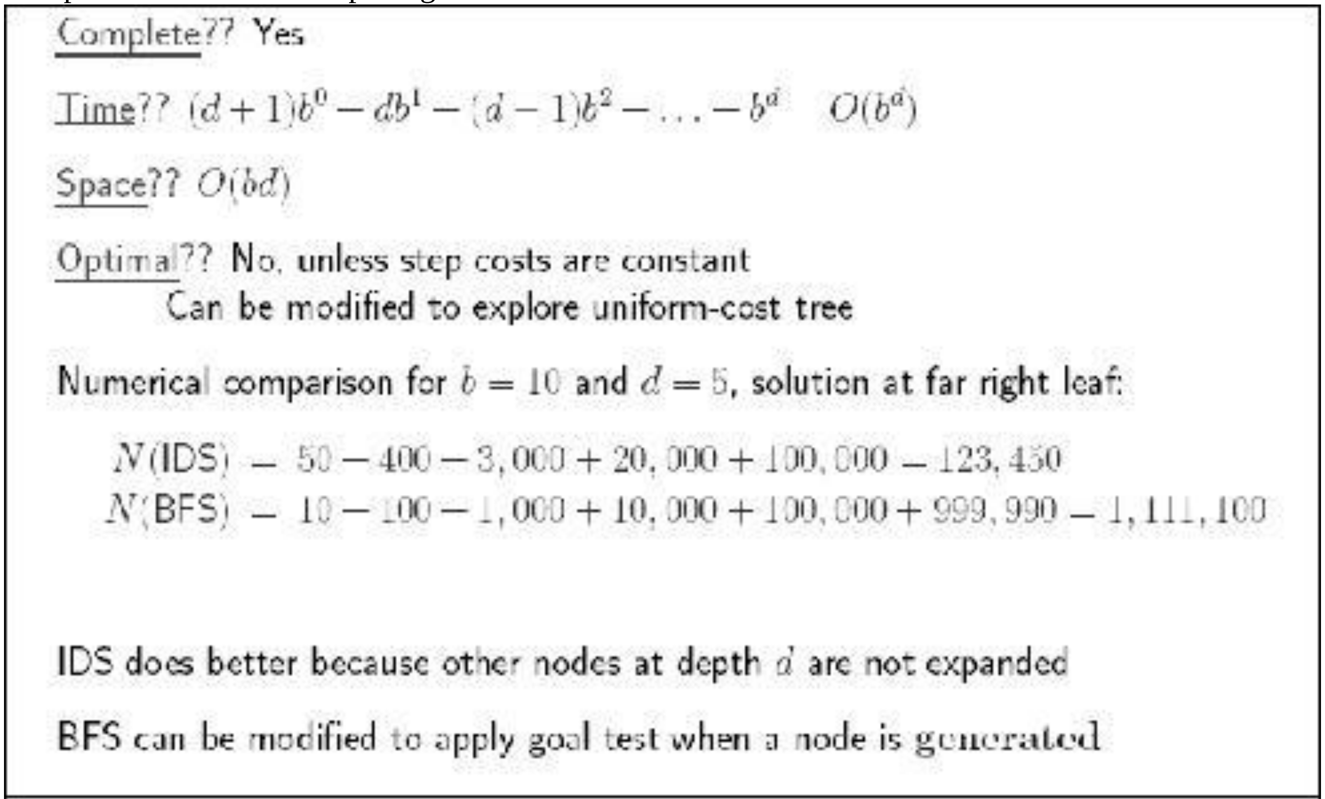


Figure 1.35

Iterative search is not as wasteful as it might seem

Properties of iterative deepening search



Complete?? Yes

Time?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space?? $O(bd)$

Optimal?? No, unless step costs are constant
Can be modified to explore uniform-cost tree

Numerical comparison for $b = 10$ and $d = 5$, solution at far right leaf:

$$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$
$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100$$

IDS does better because other nodes at depth d are not expanded

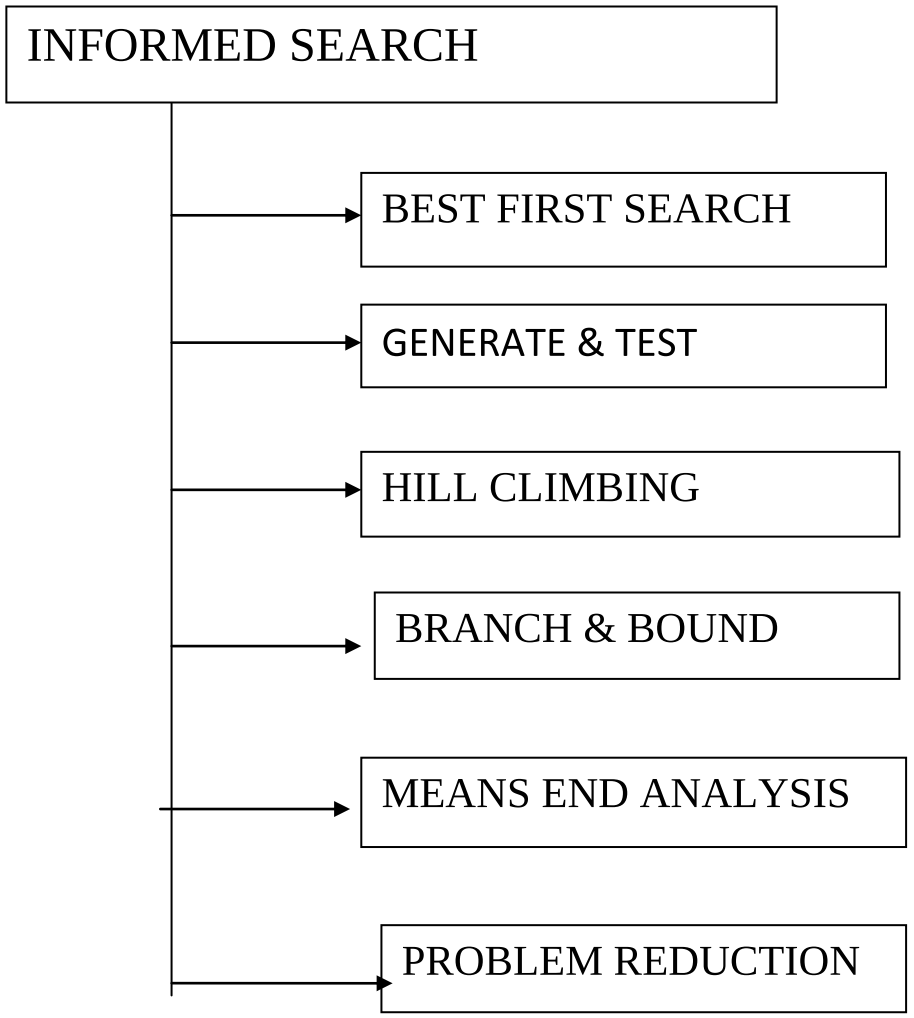
BFS can be modified to apply goal test when a node is generated

Figure 1.36

INFORMED SEARCH AND EXPLORATION

Informed(Heuristic) Search Strategies

Informed search strategy is one that uses problem-specific knowledge beyond the definition of the problem itself. It can find solutions more efficiently than uninformed strategy.



Best-first search

Best-first search is an instance of general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an **evaluation function** f(n). The node with lowest evaluation is selected for expansion,because the evaluation measures the distance to the goal.

This can be implemented using a priority-queue,a data structure that will maintain the fringe in ascending order of f-values.

. Heuristic functions

A **heuristic function** or simply a **heuristic** is a function that ranks alternatives in various search algorithms at each branching step basing on an available information in order to make a decision which branch is to be followed during a search.

The key component of Best-first search algorithm is a **heuristic function**,denoted by h(n):

h(n) = extimated cost of the **cheapest path** from node n to a **goal node**.

For example,in Romania,one might estimate the cost of the cheapest path from Arad to Bucharest via a **straight-line distance** from Arad to Bucharest(Figure 2.1).

Heuristic function are the most common form in which additional knowledge is imparted to the search algorithm.

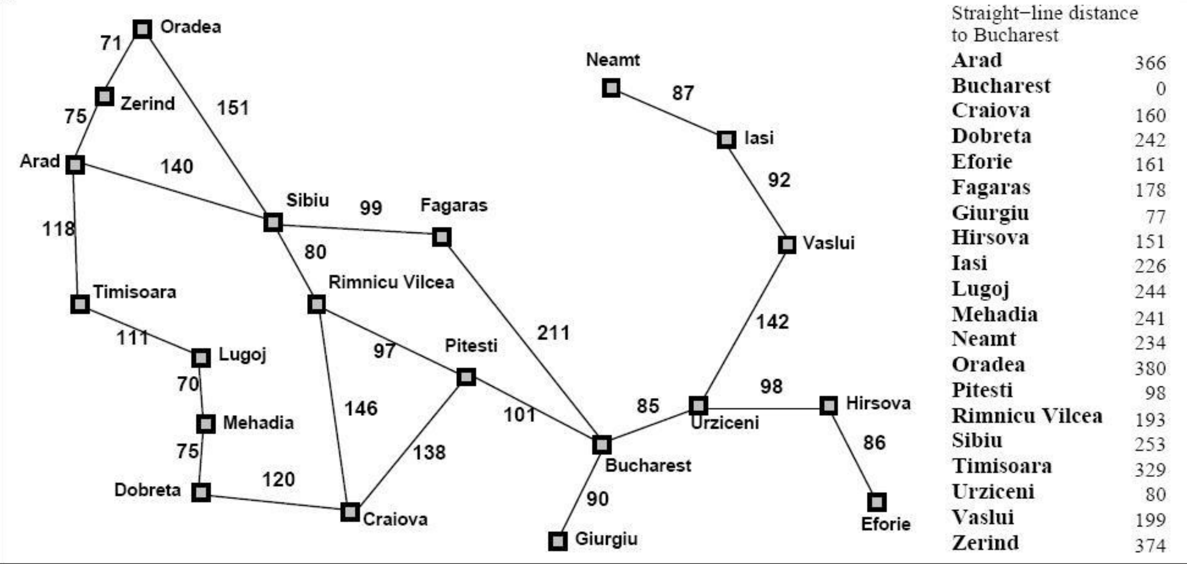
Greedy Best-first search

Greedy best-first search tries to expand the node that is closest to the goal,on the grounds that this is likely to a solution quickly.

It evaluates the nodes by using the heuristic function f(n) = h(n).

Taking the example of **Route-finding problems** in Romania , the goal is to reach Bucharest starting from the city Arad. We need to know the straight-line distances to Bucharest from various cities as shown in Figure 2.1. For example, the initial state is In(Arad) ,and the straight line distance heuristic hSLD(In(Arad)) is found to be 366.

Using the **straight-line distance** heuristic **hSLD ,**the goal state can be reached faster.



Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Figure 2.1 Values of hSLD - straight line distances to Bucharest

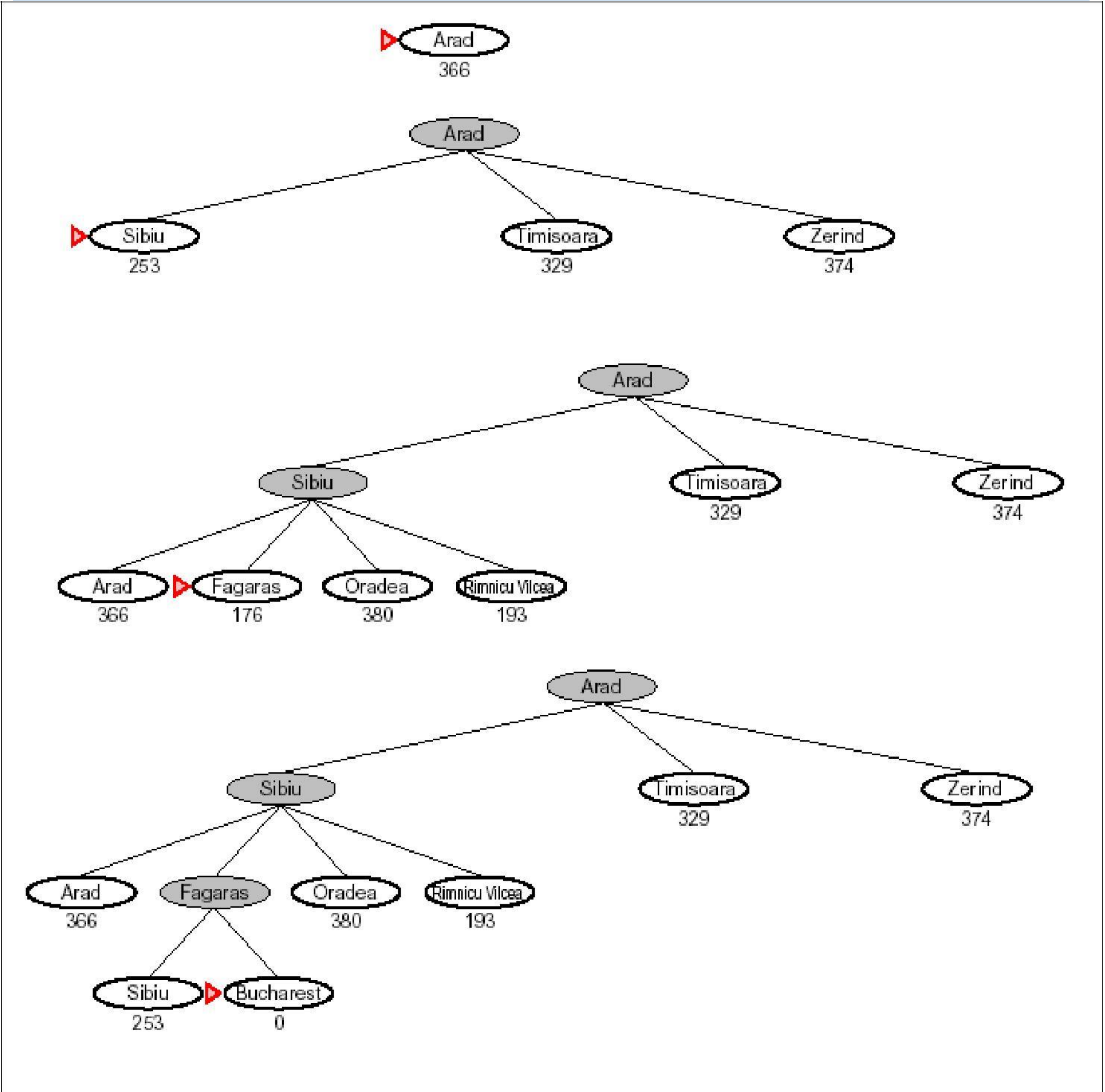


Figure 2.2 stages in greedy best-first search for Bucharest using straight-line distance heuristic hSLD. Nodes are labeled with their h-values.

Figure 2.2 shows the progress of greedy best-first search using hSLD to find a path from Arad to Bucharest. The first node to be expanded from Arad will be Sibiu,because it is closer to Bucharest than either Zerind or Timisoara. The next node to be expanded will be Fagaras,because it is closest. Fagaras in turn generates Bucharest,which is the goal.

Properties of greedy search

o **Complete??** No–can get stuck in loops, e.g.,
Iasi ! Neamt ! Iasi ! Neamt !
Complete in finite space with repeated-state checking
o **Time??** O(bm), but a good heuristic can give dramatic improvement o
Space?? O(bm)—keeps all nodes in memory
o **Optimal??** No

Greedy best-first search is not optimal,and it is incomplete.
The worst-case time and space complexity is O(b^m),where m is the maximum depth of the search space.

A* Search

A* Search is the most widely used form of best-first search. The evaluation function f(n) is obtained by combining

(1) **g(n) =** the cost to reach the node,and
(2) **h(n) =** the cost to get from the node to the **goal** :

$$f(n) = g(n) + h(n).$$

A* Search is both optimal and complete. A* is optimal if h(n) is an admissible heuristic. The obvious example of admissible heuristic is the straight-line distance hSLD. It cannot be an overestimate.
A* Search is optimal if h(n) is an admissible heuristic – that is,provided that h(n) never overestimates the cost to reach the goal.

An obvious example of an admissible heuristic is the straight-line distance hSLD that we used in getting to Bucharest. The progress of an A* tree search for Bucharest is shown in Figure 2.2.

The values of 'g ' are computed from the step costs shown in the Romania map(figure 2.1). Also the values of hSLD are given in Figure 2.1.

Recursive Best-first Search(RBFS)

Recursive best-first search is a simple recursive algorithm that attempts to mimic the operation of standard best-first search,but using only linear space. The algorithm is shown in figure 2.4.

Its structure is similar to that of recursive depth-first search,but rather than continuing indefinitely down the current path,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 recursion unwinds back to the alternative path. As the recursion unwinds,RBFS replaces the f-value of each node along the path with the best f-value of its children.
Figure 2.5 shows how RBFS reaches Bucharest.

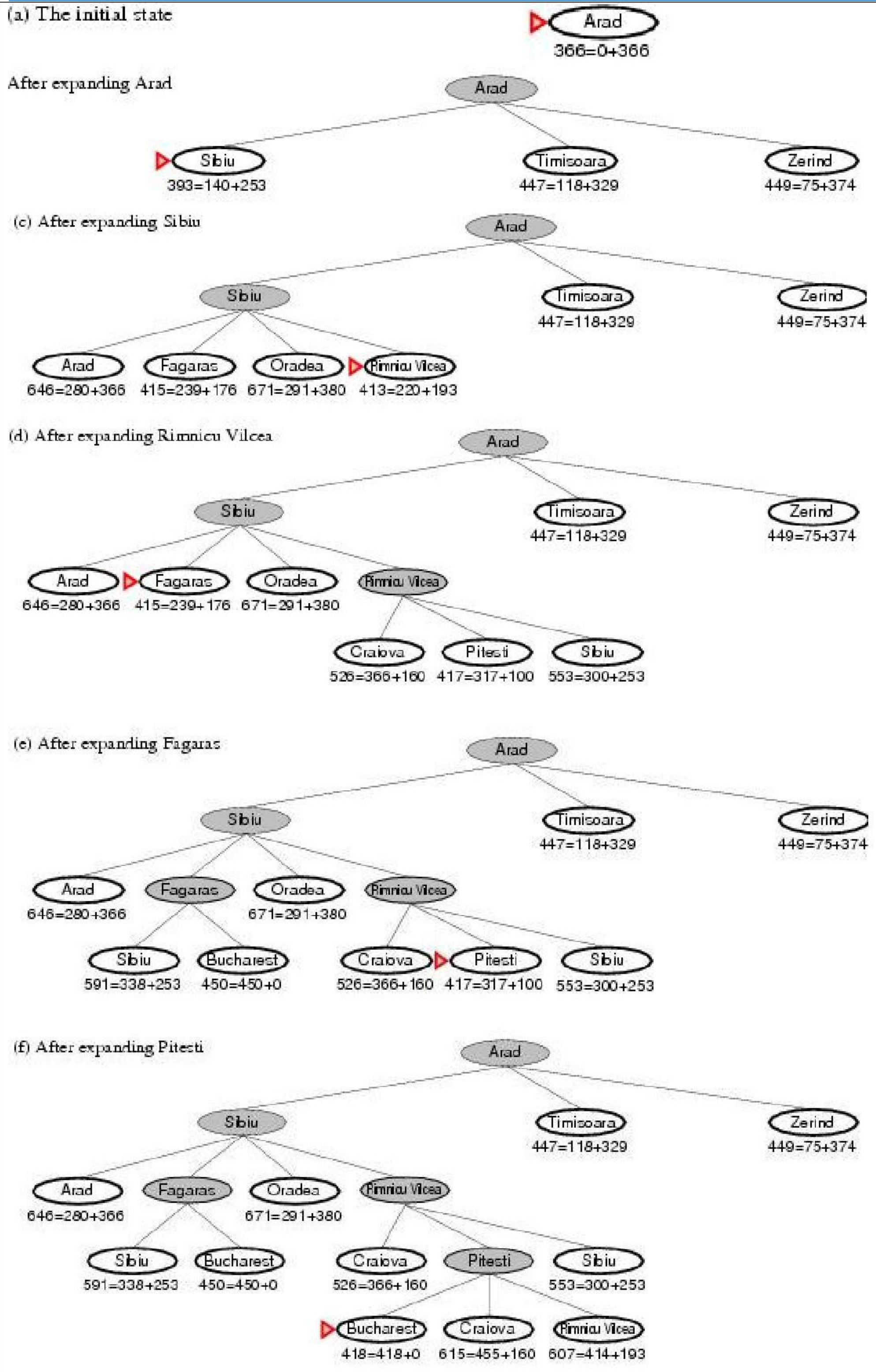


Figure 2.3 Stages in A* Search for Bucharest. Nodes are labeled with f = g + h . The h-values are the straight-line distances to Bucharest taken from figure 2.1

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) return a solution or failure
    return RFBS(problem,MAKE-NODE(INITIAL-STATE[problem]),∞)

function RFBS( problem, node, f_limit) return a solution or failure and a new f-cost limit
    if GOAL-TEST[problem](STATE[node]) then return node
    successors □ EXPAND(node, problem)
    if successors is empty then return failure, ∞
    for each s in successors do
        f [s] □ max(g(s) + h(s), f [node])
    repeat
        best □ the lowest f-value node in successors
        if f [best] > f_limit then return failure, f [best]
        alternative □ the second lowest f-value among successors
        result, f [best] □ RBFS(problem, best, min(f_limit, alternative))
        if result □ failure then return result
```

Figure 2.4 The algorithm for recursive best-first search

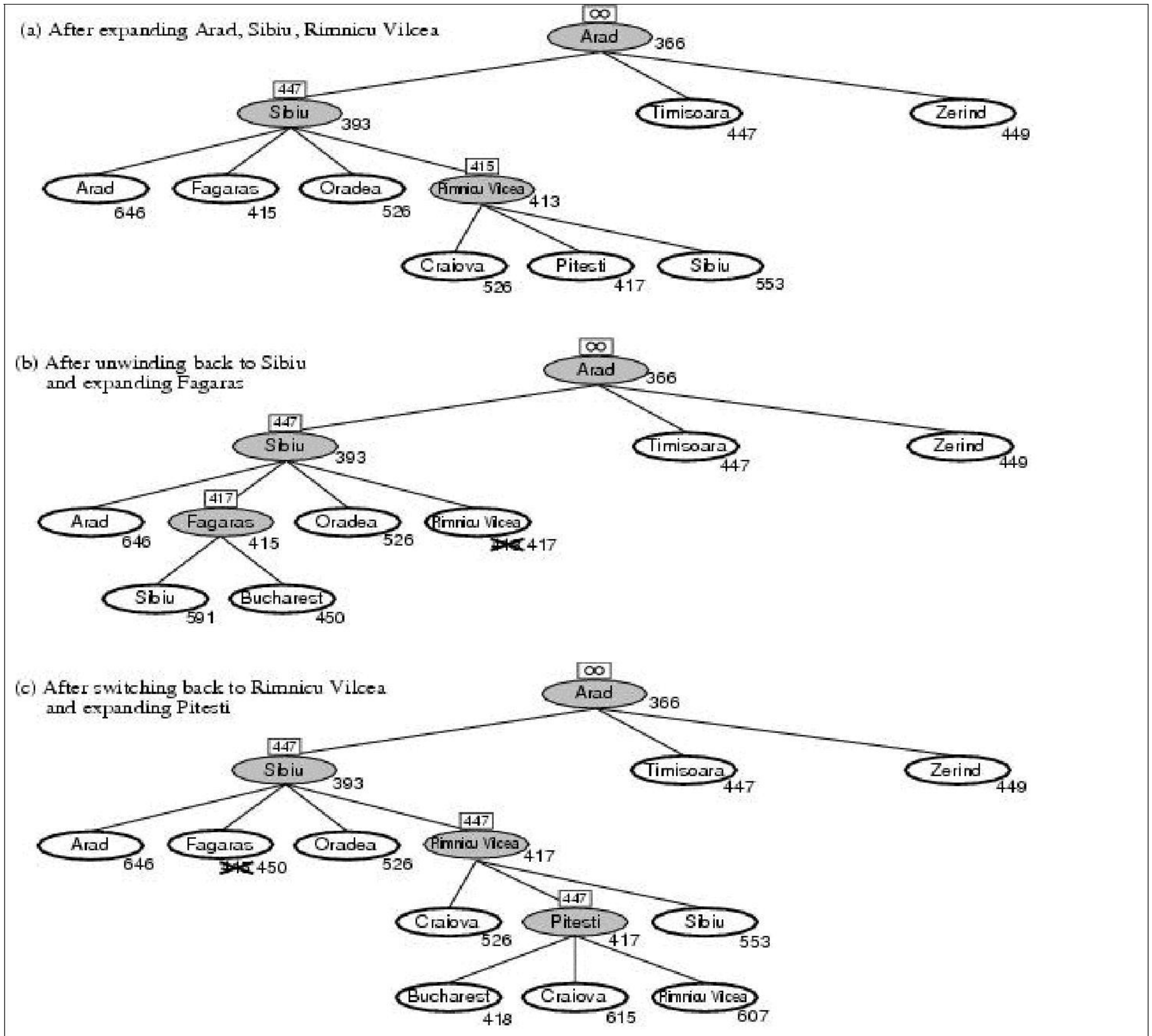


Figure 2.5 Stages in an RBFS search for the shortest route to Bucharest. The f-limit value for each recursive call is shown on top of each current node. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras).

(b) The recursion unwinds and the best leaf value of the forgotten subtree (417) is backed up to Rimnicu Vilcea;then Fagaras is expanded,revealing a best leaf value of 450.

(c) The recursion unwinds and the best leaf value of the forgotten subtree (450) is backed upto Fagaras; then Rimni Vicea is expanded. This time because the best alternative path(through Timisoara) costs atleast 447,the expansion continues to Bucharest

RBFS **Evaluation** :.

RBFS is a bit more efficient than IDA*

– Still excessive node generation (mind changes)

Like A*, optimal if *h(n)* is admissible

Space complexity is *O(bd)*.

– IDA* retains only one single number (the current f-cost limit)

Time complexity difficult to characterize

– Depends on accuracy if h(n) and how often best path changes.

IDA* en RBFS suffer from ***too little*** memory.

Heuristic Functions

A **heuristic function** or simply a heuristic is a function that ranks alternatives in various search algorithms at each branching step basing on an available information in order to make a decision which branch is to be followed during a search

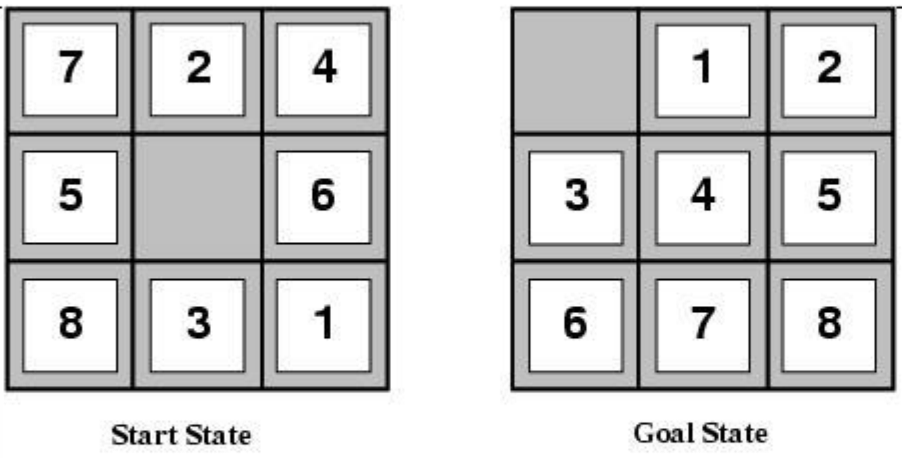


Figure 2.6 A typical instance of the 8-puzzle.
The solution is 26 steps long.

The 8-puzzle

The 8-puzzle is an example of Heuristic search problem. The object of the puzzle is to slide the tiles horizontally or vertically into the empty space until the configuration matches the goal configuration(Figure 2.6)

The average cost for a randomly generated 8-puzzle instance is about 22 steps. The branching factor is about 3.(When the empty tile is in the middle,there are four possible moves;when it is in the corner there are two;and when it is along an edge there are three). This means that an exhaustive search to depth 22 would look at about 3^{22} approximately = 3.1 X 10^{10} states.

By keeping track of repeated states,we could cut this down by a factor of about 170,000,because there are only 9!/2 = 181,440 distinct states that are reachable. This is a manageable number ,but the corresponding number for the 15-puzzle is roughly 10^{13}.

If we want to find the shortest solutions by using A^*,we need a heuristic function that never overestimates the number of steps to the goal.

The two commonly used heuristic functions for the 15-puzzle are :

(1) h1 = the number of misplaced tiles.

For figure 2.6 ,all of the eight tiles are out of position,so the start state would have h1 = 8. h1 is an admissible heuristic.

(2) h2 = the sum of the distances of the tiles from their goal positions. This is called **the city block distance** or **Manhattan distance.**

h2 is admissible ,because all any move can do is move one tile one step closer to the goal. Tiles 1 to 8 in start state give a Manhattan distance of

h2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18.

Neither of these overestimates the true solution cost ,which is 26.

The Effective Branching factor

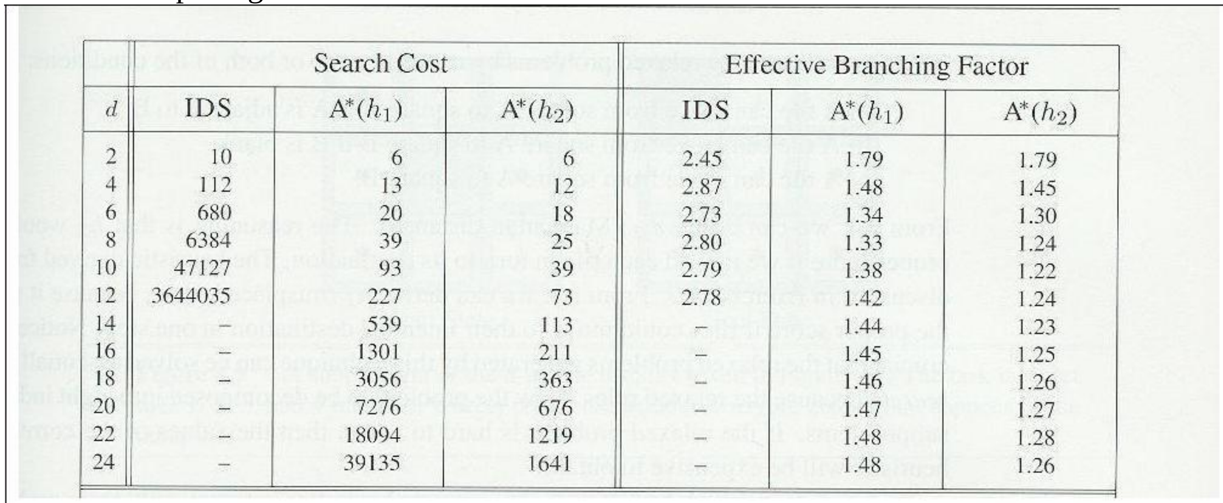
One way to characterize the **quality of a heuristic** is the **effective branching factor b*.** If the total number of nodes generated by A* for a particular problem is **N**,and the **solution depth** is **d,**then b^* is the branching factor that a uniform tree of depth d would have to have in order to contain N+1 nodes. Thus,

$$N + 1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d$$

For example,if A^* finds a solution at depth 5 using 52 nodes,then effective branching factor is 1.92. A well designed heuristic would have a value of b^* close to 1,allowing failru large problems to be solved.

To test the heuristic functions h1 and h2,1200 random problems were generated with solution lengths from 2 to 24 and solved them with iterative deepening search and with A^* search using both h1 and h2. Figure 2.7 gives the averaghe number of nodes expanded by each strategy and the effective branching factor.

The results suggest that h2 is better than h1,and is far better than using iterative deepening search. For a solution length of 14,A^* with h2 is 30,000 times more efficient than uninformed iterative deepening search.



	Search Cost			Effective Branching Factor		
d	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Figure 2.7 Comparison of search costs and effective branching factors for the ITERATIVE-DEEPENING-SEARCH and A^* Algorithms with h1,and h2. Data are average over 100 instances of the 8-puzzle,for various solution lengths.

Inventing admissible heuristic

functions Relaxed problems

- A problem with fewer restrictions on the actions is called a ***relaxed problem***
- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem
- If the rules of the 8-puzzle are relaxed so that a tile can move *anywhere*, then *h1(n)* gives the shortest solution
- If the rules are relaxed so that a tile can move to *any adjacent square,* then *h2(n)* gives the shortest solution

LOCAL SEARCH ALGORITHMS AND OPTIMIZATION PROBLEMS

- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution
- For example,in the 8-queens problem,what matters is the final configuration of queens,not the order in which they are added.

- In such cases, we can **use local search algorithms.** They operate using a **single current state**(rather than multiple paths) and generally move only to neighbors of that state.
- The important applications of these class of problems are (a) integrated-circuit design,(b)Factory-floor layout,(c) job-shop scheduling,(d)automatic programming,(e)telecommunications network optimization,(f)Vehicle routing,and (g) portfolio management.

Key advantages of Local Search Algorithms

(1) They use very little memory – usually a constant amount; and

(2) they can often find reasonable solutions in large or infinite(continuous) state spaces for which systematic algorithms are unsuitable.

OPTIMIZATION PROBLEMS

Inaddition to finding goals,local search algorithms are useful for solving pure **optimization problems**,in which the aim is to find the **best state** according to an **objective function**.

State Space Landscape

To understand local search,it is better explained using **state space landscape** as shown in figure 2.8.

A landscape has both "**location**" (defined by the state) and "**elevation**"(defined by the value of the heuristic cost function or objective function).

If elevation corresponds to **cost**,then the aim is to find the **lowest valley** – a **global minimum**; if elevation corresponds to an **objective function**,then the aim is to find the **highest peak** – a **global maximum.**

Local search algorithms explore this landscape. A complete local search algorithm always finds a **goal** if one exists; an **optimal** algorithm always finds a **global minimum/maximum**.

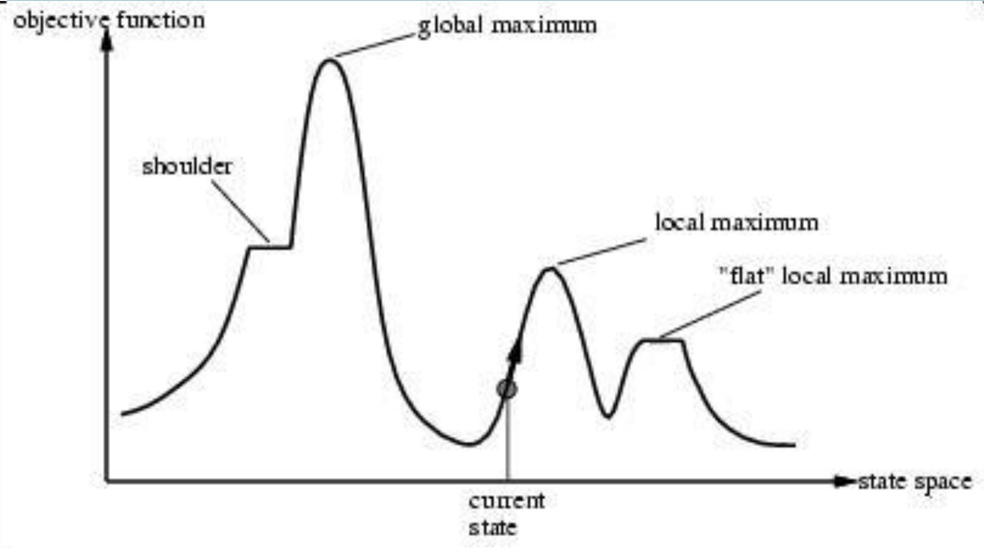


Figure 2.8 A one dimensional **state space landscape** in which elevation corresponds to the **objective function**. The aim is to find the global maximum. Hill climbing search modifies the current state to try to improve it ,as shown by the arrow. The various topographic features are defined in the text

Hill-climbing search

The **hill-climbing** search algorithm as shown in figure 2.9, is simply a loop that continually moves in the direction of increasing value – that is,**uphill**. It terminates when it reaches a "**peak**" where no neighbor has a higher value.

```
function HILL-CLIMBING( problem) return a state that is a local
    maximum input: problem, a problem
    local variables: current, a node.
                     neighbor, a
                     node.

    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        neighbor ← a highest valued successor of current
        if VALUE [neighbor] ≤ VALUE[current] then return STATE[current]
        current ← neighbor
```

Figure 2.9 The hill-climbing search algorithm (steepest ascent version),which is the most basic local search technique. At each step the current node is replaced by the best neighbor;the neighbor with the highest VALUE. If the heuristic cost estimate h is used,we could find the neighbor with the lowest h.

Hill-climbing is sometimes called greedy local search because it grabs a good neighbor state without thinking ahead about where to go next. Greedy algorithms often perform quite well.

Problems with hill-climbing

Hill-climbing often gets stuck for the following reasons :

- **Local maxima** : a local maximum is a peak that is higher than each of its neighboring states,but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upwards towards the peak,but will then be stuck with nowhere else to go
- **Ridges** : A ridge is shown in Figure 2.10. Ridges results in a sequence of local maxima that is very difficult for greedy algorithms to navigate.
- **Plateaux** : A plateau is an area of the state space landscape where the evaluation function is flat. It can be a flat local maximum,from which no uphill exit exists,or a shoulder,from which it is possible to make progress.

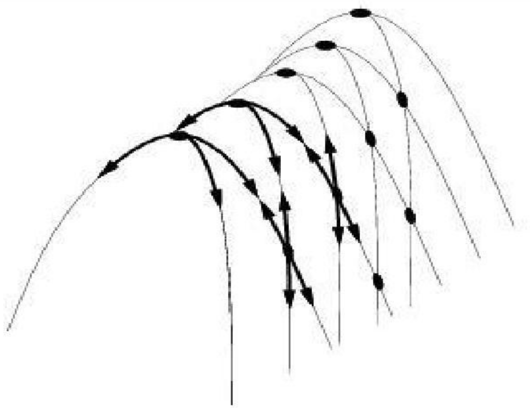


Figure 2.10 Illustration of why ridges cause difficulties for hill-climbing. The grid of states(dark circles) is superimposed on a ridge rising from left to right,creating a sequence of local maxima that are not directly connected to each other. From each local maximum,all th available options point downhill.

Hill-climbing variations

- **Stochastic hill-climbing**
  - Random selection among the uphill moves.
  - The selection probability can vary with the steepness of the uphill move.
- **First-choice hill-climbing**
  - cfr. stochastic hill climbing by generating successors randomly until a better one is found.
- **Random-restart hill-climbing**
  - Tries to avoid getting stuck in local maxima.

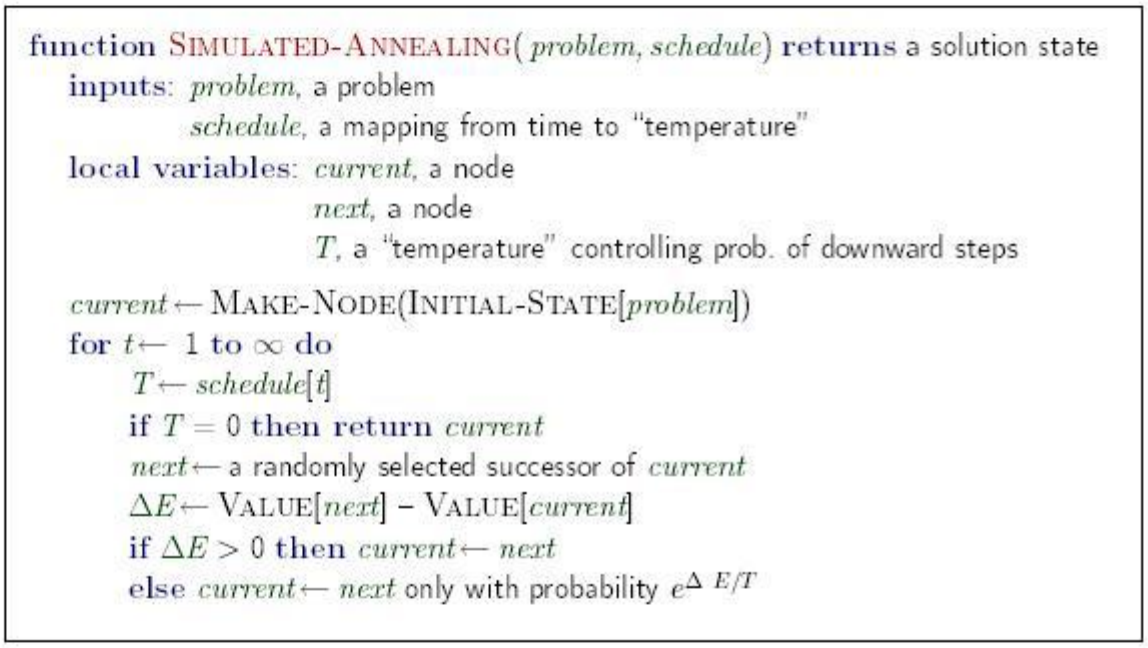
Simulated annealing search

A hill-climbing algorithm that never makes “downhill” moves towards states with lower value(or higher cost) is guaranteed to be incomplete, because it can stuck on a local maximum. In contrast, a purely random walk –that is, moving to a successor chosen uniformly at random from the set of successors – is complete, but extremely inefficient.

Simulated annealing is an algorithm that combines hill-climbing with a random walk in someway that yields both efficiency and completeness.

Figure 2.11 shows simulated annealing algorithm. It is quite similar to hill climbing. Instead of picking the best move,however,it picks the random move. If the move improves the situation, it is always accepted. Otherwise,the algorithm accepts the move with some probability less than 1. The probability decreases exponentially with the “badness” of the move – the amount E by which the evaluation is worsened.

Simulated annealing was first used extensively to solve VLSI layout problems in the early 1980s. It has been applied widely to factory scheduling and other large-scale optimization tasks.



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    inputs: problem, a problem
            schedule, a mapping from time to "temperature"
    local variables: current, a node
                     next, a node
                     T, a "temperature" controlling prob. of downward steps

    current ← MAKE-NODE(INITIAL-STATE[problem])
    for t ← 1 to $\infty$ do
        T ← schedule[t]
        if T = 0 then return current
        next ← a randomly selected successor of current
        $\Delta E$ ← VALUE[next] – VALUE[current]
        if $\Delta E > 0$ then current ← next
        else current ← next only with probability $e^{\Delta E/T}$
```

Figure 2.11 The simulated annealing search algorithm,a version of stochastic hill climbing where some downhill moves are allowed.