
CS8691 –ARTIFICIAL INTELLIGENCE

VI SEMESTER CSE

2017 Regulations

UNIT-II

(2) SEARCHING TECHNIQUES

UNIT II PROBLEM SOLVING METHODS 9

Problem solving Methods - Search Strategies- Uninformed - Informed - Heuristics - Local Search Algorithms and Optimization Problems - Searching with Partial Observations – Constraint Satisfaction Problems – Constraint Propagation - Backtracking Search - Game Playing – Optimal Decisions in Games – Alpha - Beta Pruning - Stochastic Games

---

2.0 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.

A well-defined problem can be described by:

- **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

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.

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.

2.0.1 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.

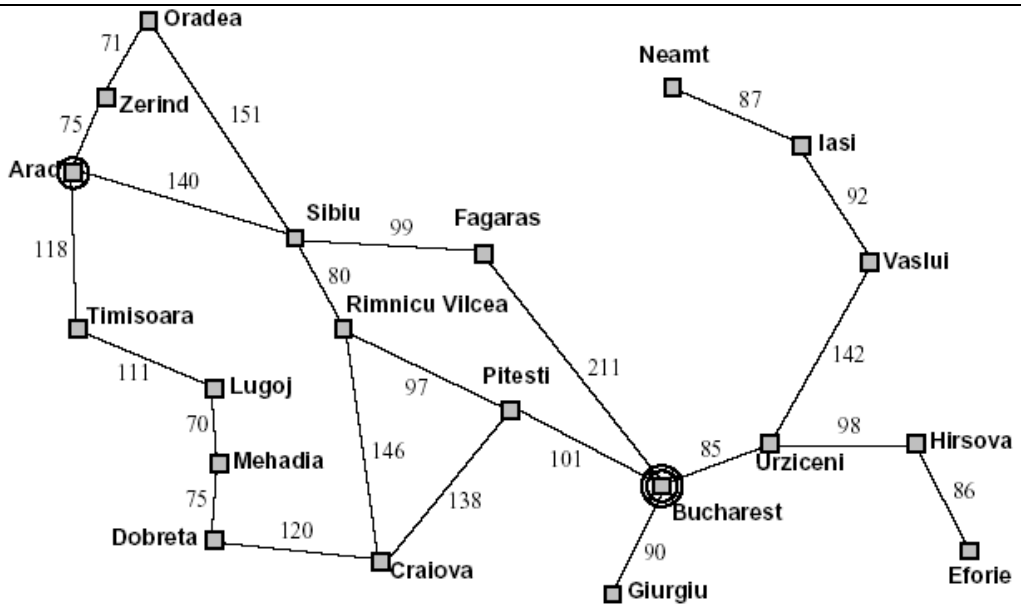


Figure 1.19 A simplified Road Map of part of Romania

2.0.2 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.

2.0.2.1 TOY PROBLEMS

Vacuum World Example

- **States**: The agent is in one of two locations.,each of which might or might not contain dirt. Thus there are $2 \times 2^2 = 8$ 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

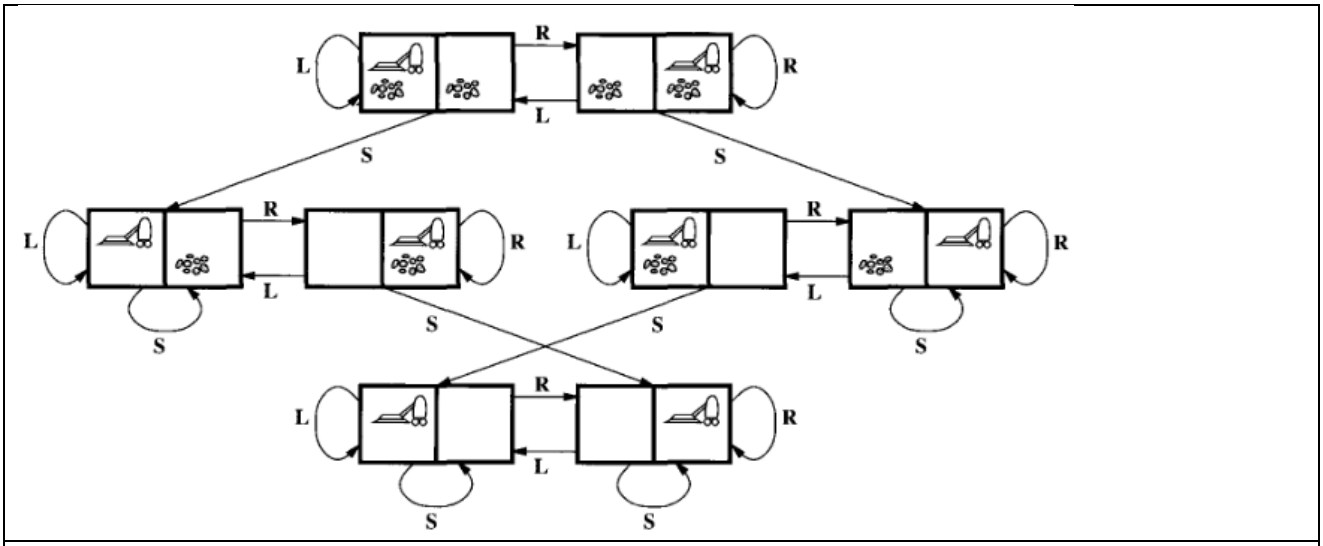


Figure 1.20 The state space for the vacuum world.
Arcs denote actions: L = Left,R = Right,S = Suck

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

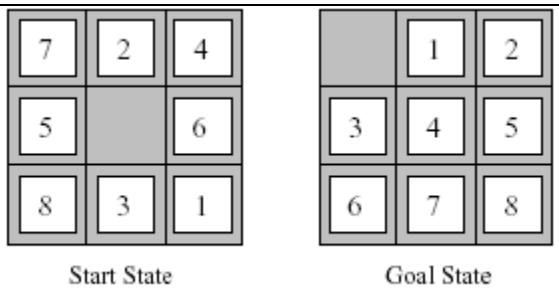


Figure 1.21 A typical instance of 8-puzzle.

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.
-

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.

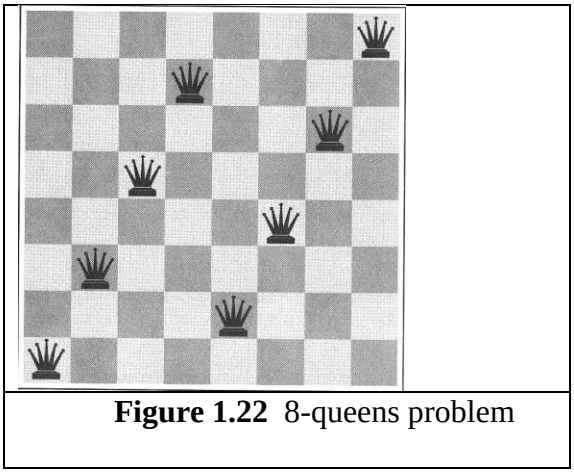


Figure 1.22 8-queens problem

The first incremental formulation one might try is the following :

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

In this formulation,we have $64.63\ldots57 = 3 \times 10^{14}$ possible sequences to investigate.

A better formulation would prohibit placing a queen in any square that is already attacked.

:

- **States** : Arrangements of n queens (0 <= n < = 8) ,one per column in the left most columns ,with no queen attacking another are states.
- **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 3×10^{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.

2.0.1 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 **:**

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

2.0.2 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.

Figure 1.23 shows some of the expansions in the search tree for finding a route from Arad to Bucharest.

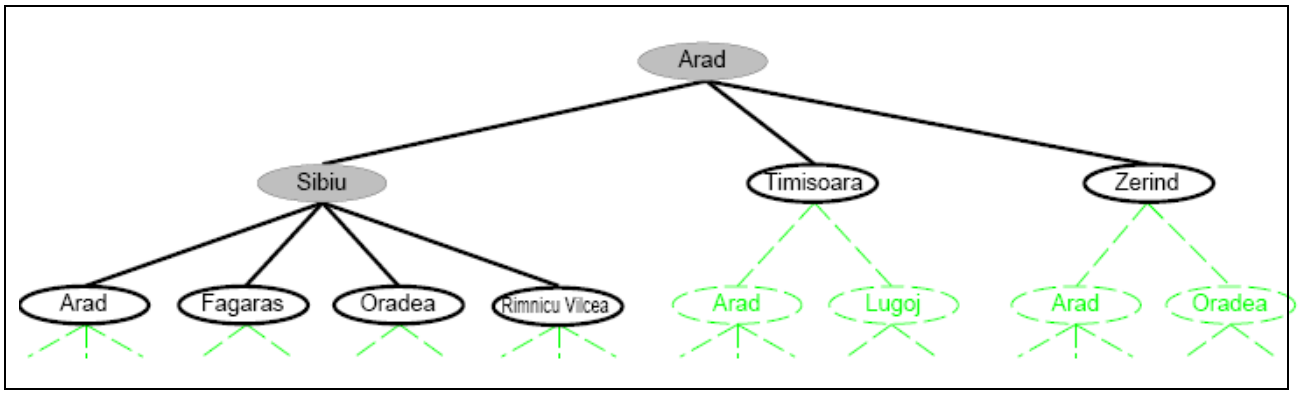


Figure 1.23 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded.; nodes that have been generated but not yet expanded are outlined in bold;nodes that have not yet been generated are shown in faint dashed line

The root of the search tree is a **search node** corresponding to the initial **state**,In(Arad). The first step is to test whether this is a **goal state**. The current state is expanded by applying the successor function to the current state,thereby generating a new set of states. In this case,we get three new states: In(Sibiu),In(Timisoara),and In(Zerind). Now we must choose which of these three possibilities to consider further. This is the essense of search- following up one option now and putting the others aside for latter,in case the first choice does not lead to a solution.
Search strategy . The general tree-search algorithm is described informally in Figure 1.24
.

Tree Search

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
      if there are no candidates for expansion then return failure
      choose a leaf node for expansion according to strategy
      if the node contains a goal state then return the corresponding solution
      else expand the node and add the resulting nodes to the search tree
```

Figure 1.24 An informal description of the general tree-search algorithm

The choice of which state to expand is determined by the **search strategy**. There are an infinite number paths in this state space ,so the search tree has an infinite number of **nodes.**
A **node** is a data structure with five components :

- STATE : a state in the state space to which the node corresponds;
- PARENT-NODE : the node in the search tree that generated this node;
- ACTION : the action that was applied to the parent to generate the node;
- PATH-COST :the cost,denoted by g(n),of the path from initial state to the node,as indicated by the parent pointers; and
- DEPTH : the number of steps along the path from the initial state.

It is important to remember the distinction between nodes and states. A node is a book keeping data structure used to represent the search tree. A state corresponds to configuration of the world.

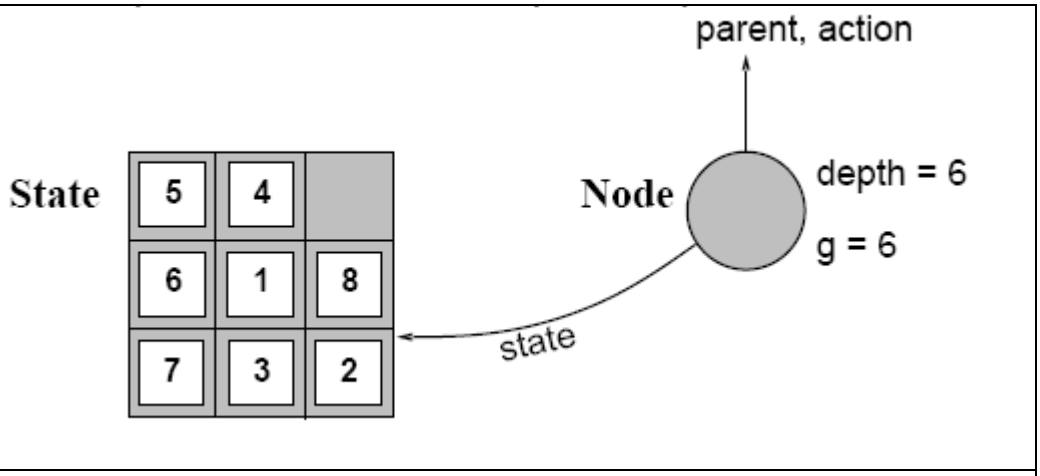


Figure 1.25 Nodes are data structures from which the search tree is constructed. Each has a parent,a state, Arrows point from child to parent.

Fringe

Fringe is a collection of nodes that have been generated but not yet been expanded. Each element of the fringe is a leaf node,that is,a node with no successors in the tree. The fringe of each tree consists of those nodes with bold outlines.
The collection of these nodes is implemented as a **queue.**
The general tree search algorithm is shown in Figure 2.9

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure

  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
     if EMPTY?(fringe) then return failure
     node ← REMOVE-FIRST(fringe)
     if GOAL-TEST[problem] applied to STATE[node] succeeds
        then return SOLUTION(node)
     fringe ← INSERT-ALL(EXPAND(node, problem), fringe)

function EXPAND(node, problem) returns a set of nodes

  successors ← the empty set
  for each ⟨action, result⟩ in SUCCESSOR-FN[problem](STATE[node]) do
     s ← a new NODE
     STATE[s] ← result
     PARENT-NODE[s] ← node
     ACTION[s] ← action
     PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
     DEPTH[s] ← DEPTH[node] + 1
     add s to successors
  return successors
```

Figure 1.26 The general Tree search algorithm

The operations specified in Figure 1.26 on a queue are as follows:

- **MAKE-QUEUE(element,…)** creates a queue with the given element(s).
- **EMPTY?(queue)** returns true only if there are no more elements in the queue.
- **FIRST(queue)** returns FIRST(queue) and removes it from the queue.
- **INSERT(element,queue)** inserts an element into the queue and returns the resulting queue.
- **INSERT-ALL(elements,queue)** inserts a set of elements into the queue and returns the resulting queue.

MEASURING PROBLEM-SOLVING PERFORMANCE

The output of problem-solving algorithm is either failure or a solution.(Some algorithms might struck in an infinite loop and never return an output.

The algorithm's performance can be measured in four ways :

- **Completeness** : Is the algorithm guaranteed to find a solution when there is one?
- **Optimality** : Does the strategy find the optimal solution
- **Time complexity** : How long does it take to find a solution?
- **Space complexity** : How much memory is needed to perform the search?

2.0.3 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.

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

2.3.4.1 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.

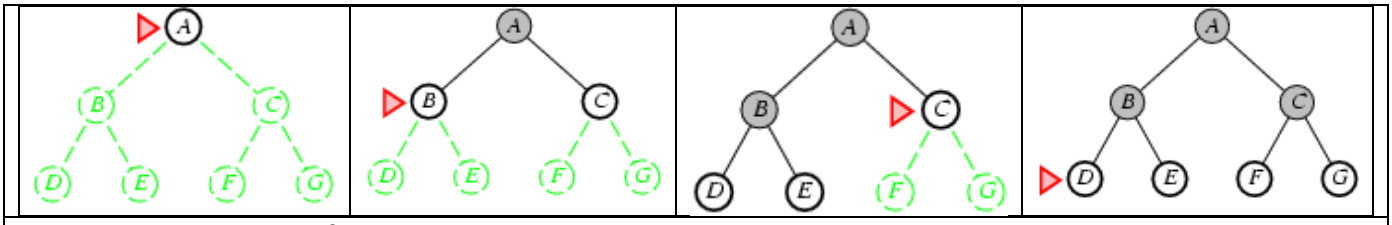


Figure 1.27 Breadth-first search on a simple binary tree. At each stage ,the node to be expanded next is indicated by a marker.

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.

Figure 1.28 Breadth-first-search properties

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

2.3.4.2 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

2.5.1.3 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.

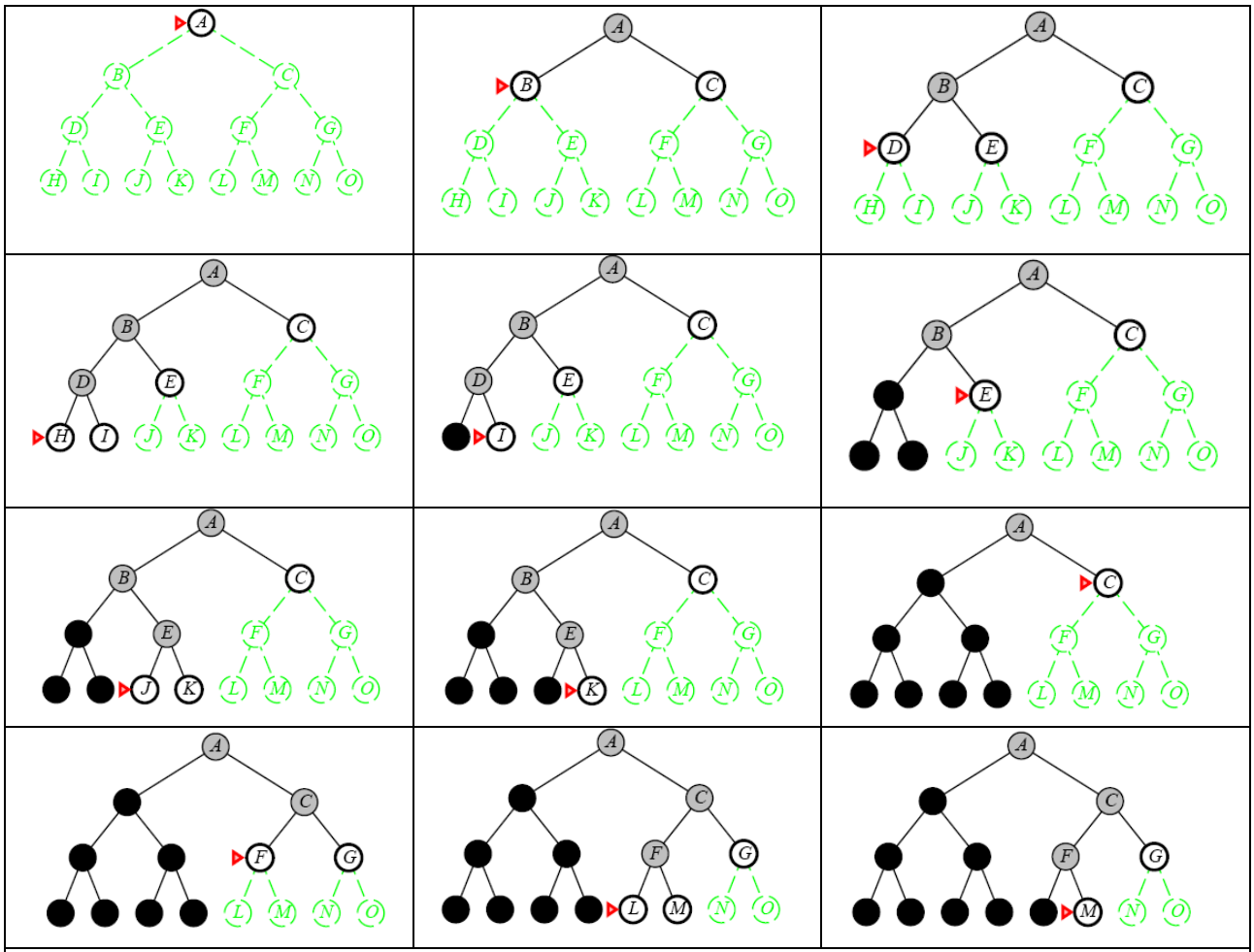


Figure 1.31 Depth-first-search on a binary tree. Nodes that have been expanded and have no descendants in the fringe can be removed from the memory;these are shown in black. Nodes at depth 3 are assumed to have no successors and M is the only goal node.

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)

2.3.4.4 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:

2.3.4.5 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.

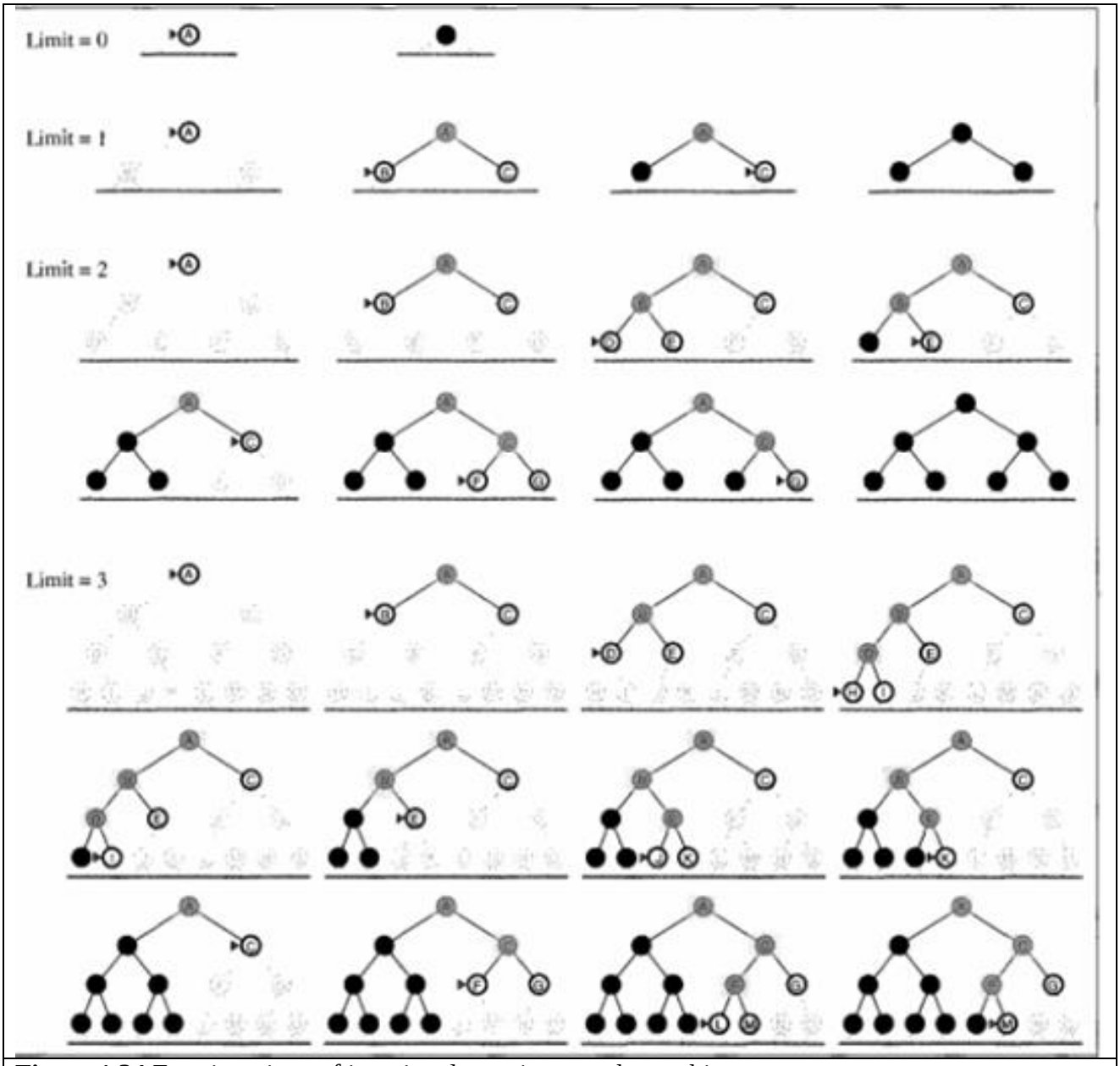


Figure 1.34 Four iterations of iterative deepening search on a binary tree

Iterative search is not as wasteful as it might seem

Iterative deepening search

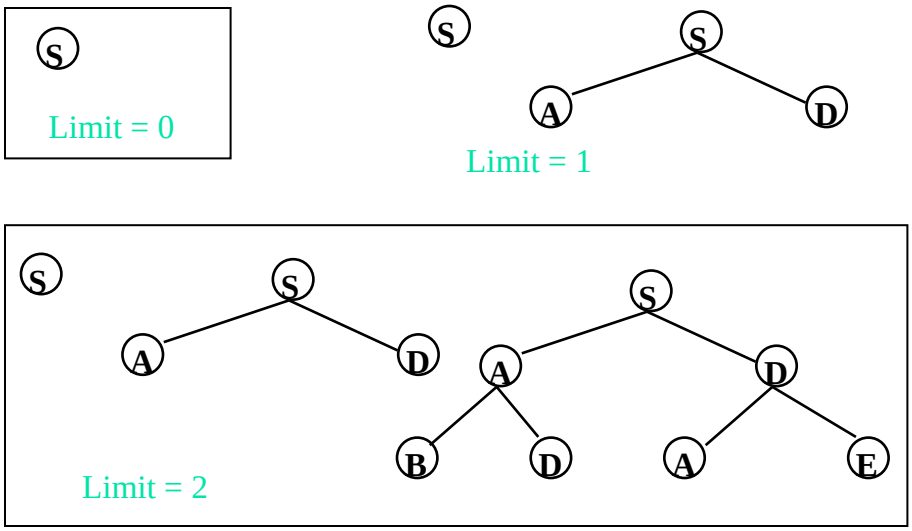


Figure 1.35

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

In general,iterative deepening is the prefered uninformed search method when there is a large search space and the depth of solution is not known.

1.3.4.6 Bidirectional Search

The idea behind bidirectional search is to run two simultaneous searches –
one forward from he initial state and
the other backward from the goal,
stopping when the two searches meet in the middle (Figure 1.37)

The motivation is that $b^{d/2} + b^{d/2}$ much less than ,or in the figure ,the area of the two small circles is less than the area of one big circle centered on the start and reaching to the goal.

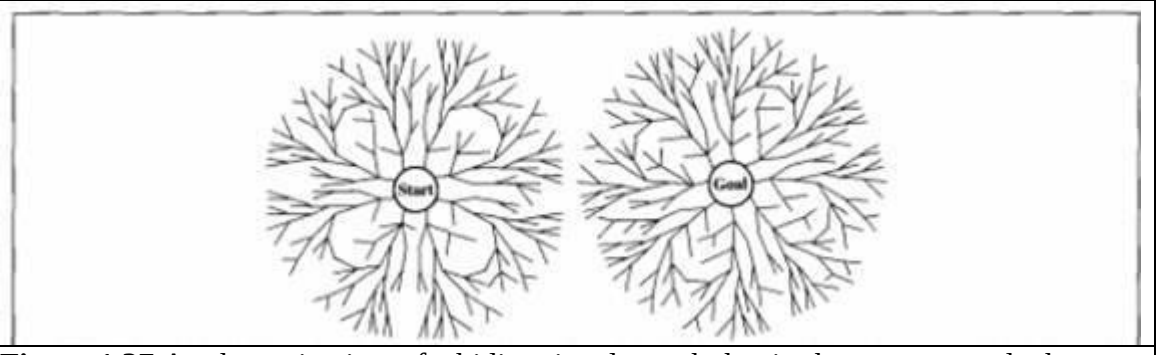


Figure 1.37 A schematic view of a bidirectional search that is about to succeed,when a Branch from the Start node meets a Branch from the goal node.

1.3.4.7 Comparing Uninformed Search Strategies

Figure 1.38 compares search strategies in terms of the four evaluation criteria .

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^{d+1})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^{d+1})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Figure 1.38 Evaluation of search strategies,b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs >= E for positive E; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

2.3.2 AVOIDING REPEATED STATES

In searching,time is wasted by expanding states that have already been encountered and expanded before. For some problems repeated states are unavoidable. The search trees for these problems are infinite. If we prune some of the repeated states,we can cut the search tree down to finite size. Considering search tree upto a fixed depth,eliminating repeated states yields an exponential reduction in search cost.

Repeated states ,can cause a solvable problem to become unsolvable if the algorithm does not detect them.

Repeated states can be the source of great inefficiency: identical sub trees will be explored many times!

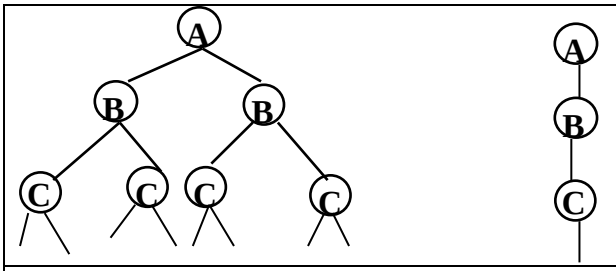


Figure 1.39

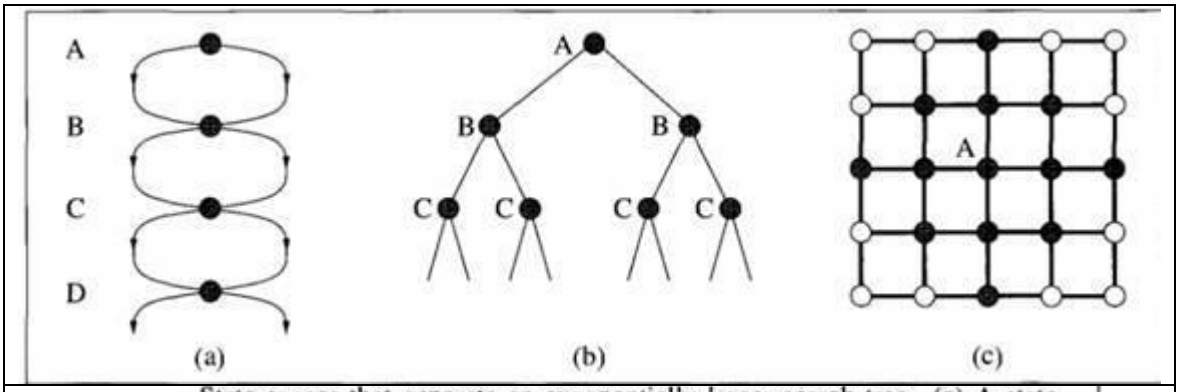


Figure 1.40 State spaces that generate an exponentially larger search tree. (a) A state space in which there are two possible actions leading from A to B, two from B to C, and so on. The state space contains $d + 1$ states, where d is the maximum depth. (b) The corresponding search tree, which has 2^d branches corresponding to the 2^d paths through the space. (c) A rectangular grid space. States within 2 steps of the initial state (A) are shown in gray.

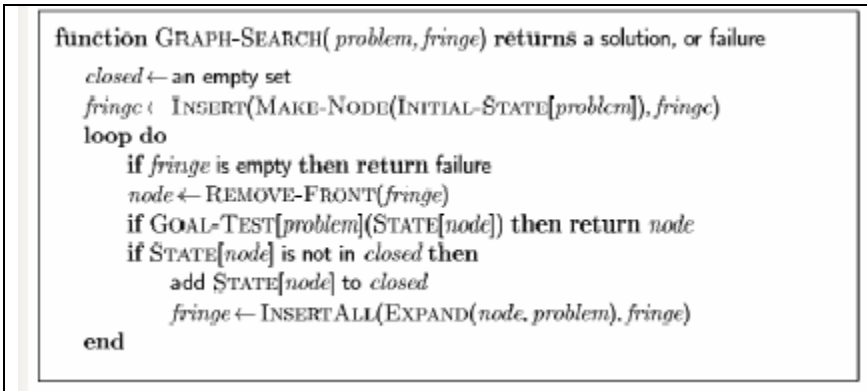


Figure 1.41 The General graph search algorithm. The set closed can be implemented with a hash table to allow efficient checking for repeated states.

Do not return to the previous state.
• Do not create paths with cycles.
• Do not generate the same state twice.
- Store states in a hash table.

- Check for repeated states.

  - Using more memory in order to check repeated state
    - *Algorithms that forget their history are doomed to repeat it.*
    - Maintain Close-List beside Open-List(fringe)

Strategies for avoiding repeated states
We can modify the general TREE-SEARCH algorithm to include the data structure called the **closed list** ,which stores every expanded node. The fringe of unexpanded nodes is called the **open list.**
If the current node matches a node on the closed list,it is discarded instead of being expanded.
The new algorithm is called GRAPH-SEARCH and much more efficient than TREE-SEARCH. The worst case time and space requirements may be much smaller than $O(b^d)$.

2.3.2 SEARCHING WITH PARTIAL INFORMATION

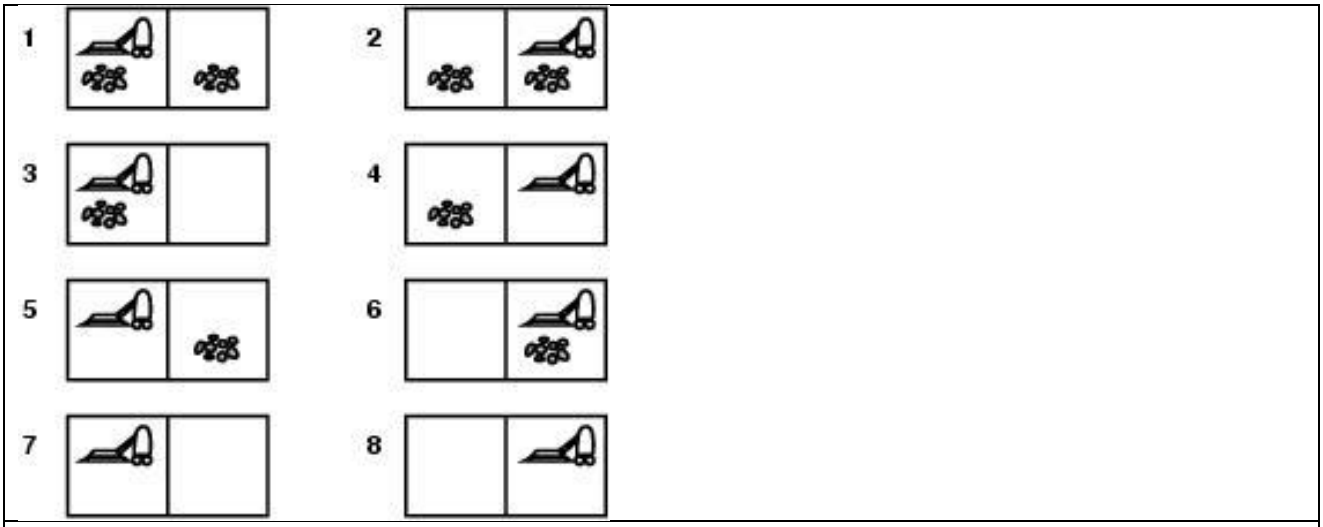
- Different types of incompleteness lead to three distinct problem types:
  - **Sensorless problems** (conformant): If the agent has no sensors at all
  - **Contingency problem**: if the environment if partially observable or if action are uncertain (adversarial)
  - **Exploration problems**: When the states and actions of the environment are unknown.

  - No sensor
  - Initial State(1,2,3,4,5,6,7,8)
  - After action [Right] the state (2,4,6,8)
  - After action [Suck] the state (4, 8)
  - After action [Left] the state (3,7)
  - After action [Suck] the state (8)

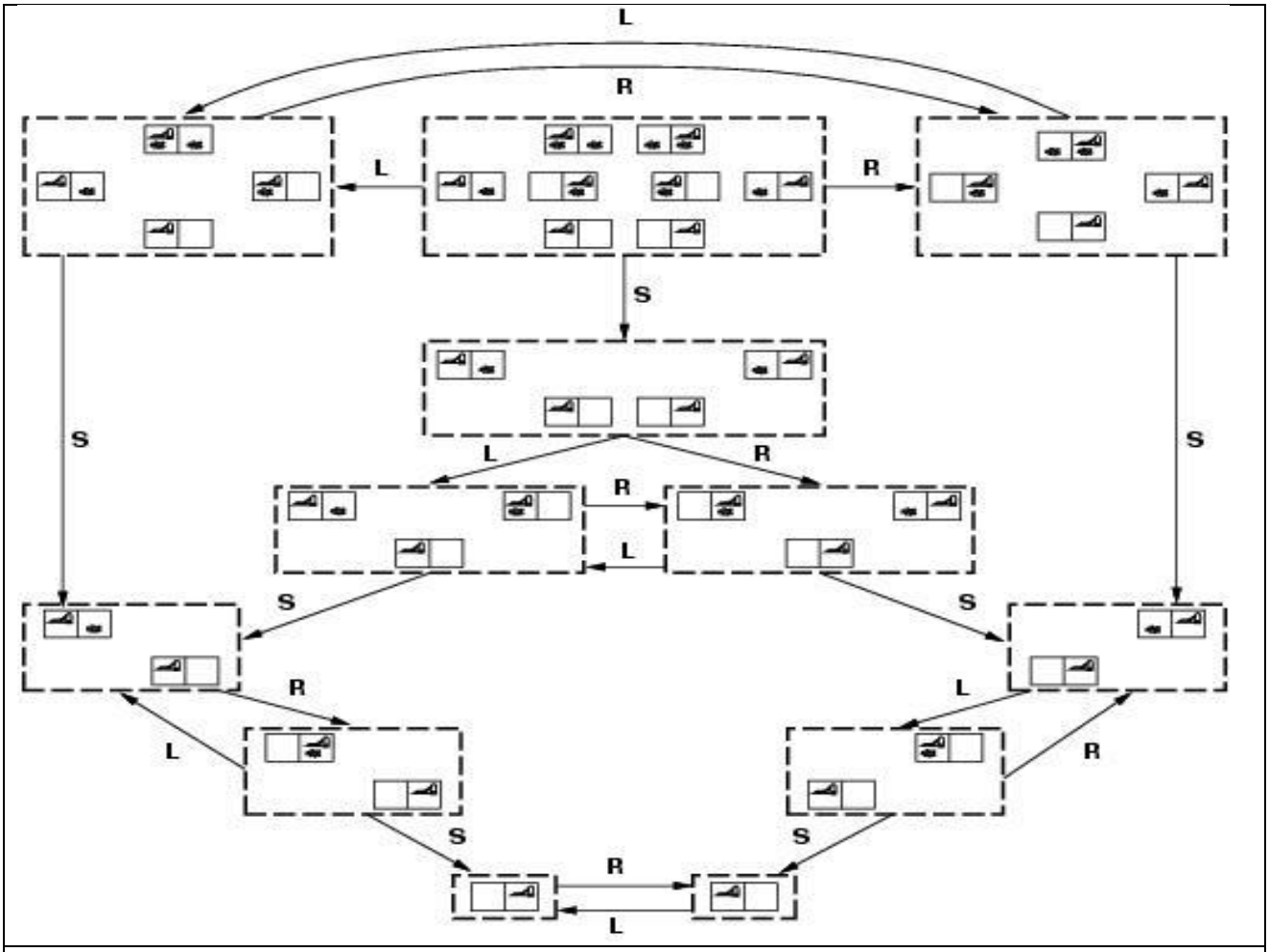
- Answer : [Right,Suck,Left,Suck] coerce the world into state 7 without any sensor
- Belief State: Such state that agent belief to be there

(SLIDE 7) Partial knowledge of states and actions:

- *sensorless or conformant problem*
  - Agent may have no idea where it is; solution (if any) is a sequence.
- *contingency problem*
  - Percepts provide *new* information about current state; solution is a tree or policy; often interleave search and execution.
  - If uncertainty is caused by actions of another agent: *adversarial problem*
- *exploration problem*
  - When states and actions of the environment are unknown.



Figure



Figure

Contingency, start in {1,3}.
Murphy's law, Suck *can* dirty a clean carpet.
Local sensing: dirt, location only.

- Percept = [L,Dirty] ={1,3}
- [Suck] = {5,7}
- [Right] ={6,8}
- [Suck] in {6}={8} (Success)
- BUT [Suck] in {8} = failure

Solution??

- Belief-state: no fixed action sequence guarantees solution

Relax requirement:

- [*Suck, Right*, if *[R,dirty]* then *Suck*]
- Select actions based on contingencies arising during execution.

Time and space complexity are always considered with respect to some measure of the problem difficulty. In theoretical computer science ,the typical measure is the size of the state space.
In AI,where the graph is represented implicitly by the initial state and successor function,the complexity is expressed in terms of three quantities:

b,the **branching factor** or maximum number of successors of any node;
d,the **depth** of the **shallowest goal node**; and
m,the **maximum length** of any path in the state space.

Search-cost - typically depends upon the time complexity but can also include the term for memory usage.
Total–cost – It combines the search-cost and the path cost of the solution found.

2.1 INFORMED SEARCH AND EXPLORATION

2.1.1 Informed(Heuristic) Search Strategies
2.1.2 Heuristic Functions
2.1.3 Local Search Algorithms and Optimization Problems
2.1.4 Local Search in Continuous Spaces
2.1.5 Online Search Agents and Unknown Environments

2.2 CONSTRAINT SATISFACTION PROBLEMS(CSP)

2.2.1 Constraint Satisfaction Problems
2.2.2 Backtracking Search for CSPs
2.2.3 The Structure of Problems

2.3 ADVERSARIAL SEARCH

2.3.1 Games
2.3.2 Optimal Decisions in Games
2.3.3 Alpha-Beta Pruning
2.3.4 Imperfect ,Real-time Decisions
2.3.5 Games that include Element of Chance

2.1 INFORMED SEARCH AND EXPLORATION

2.1.1 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.

2.1.2. 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.

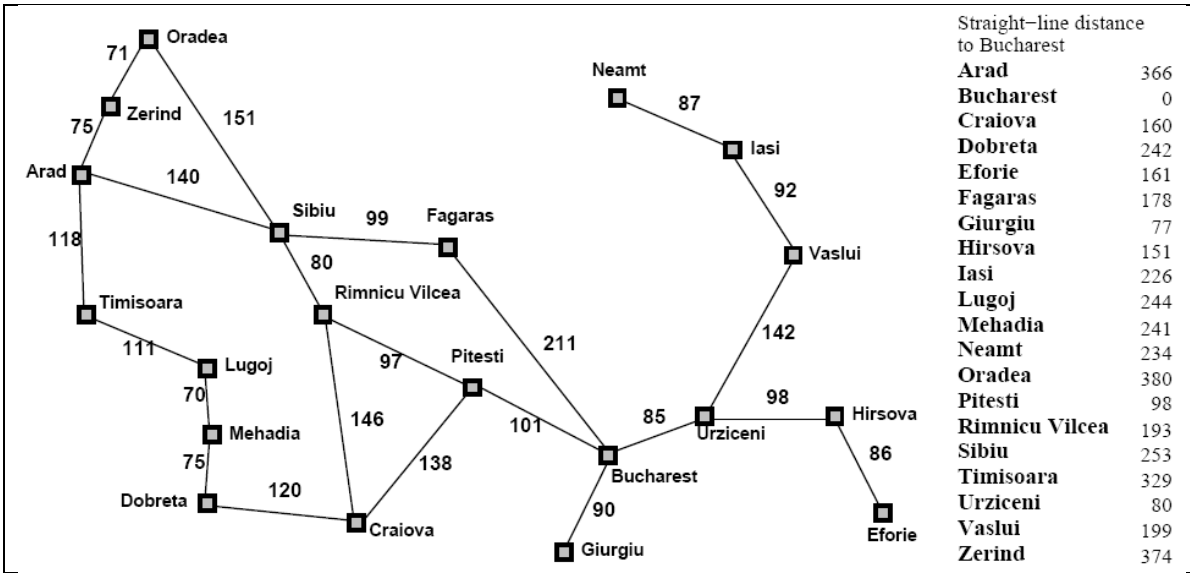
The key component of Best-first search algorithm is a **heuristic function**,denoted by h(n):

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 h_{SLD}(In(Arad)) is found to be 366.
Using the **straight-line distance** heuristic $\mathbf{h_{SLD}}$ **,**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 h_{SLD} - straight line distances to Bucharest

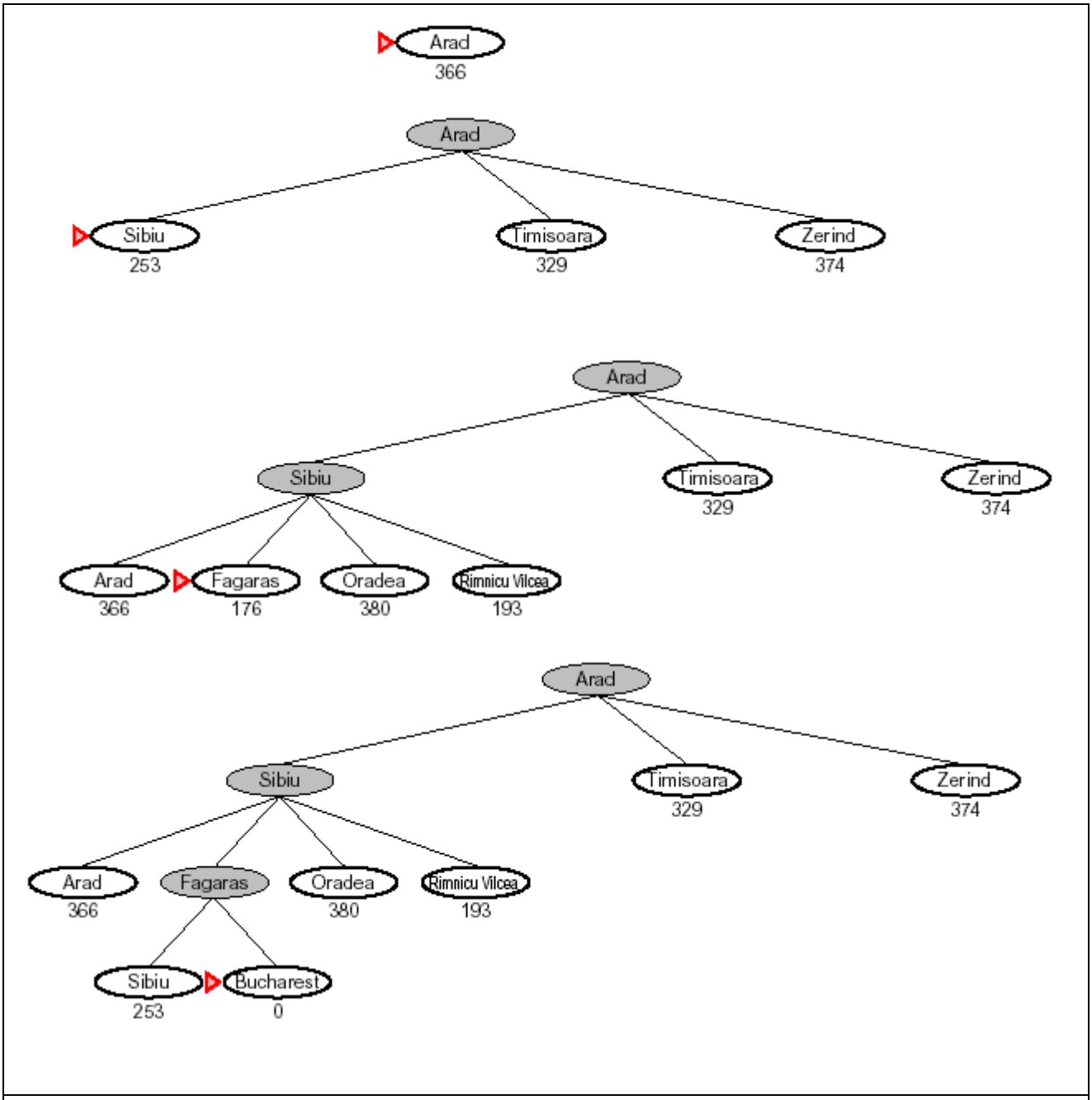


Figure 2.2 stages in greedy best-first search for Bucharest using straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

Figure 2.2 shows the progress of greedy best-first search using h_{SLD} 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

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

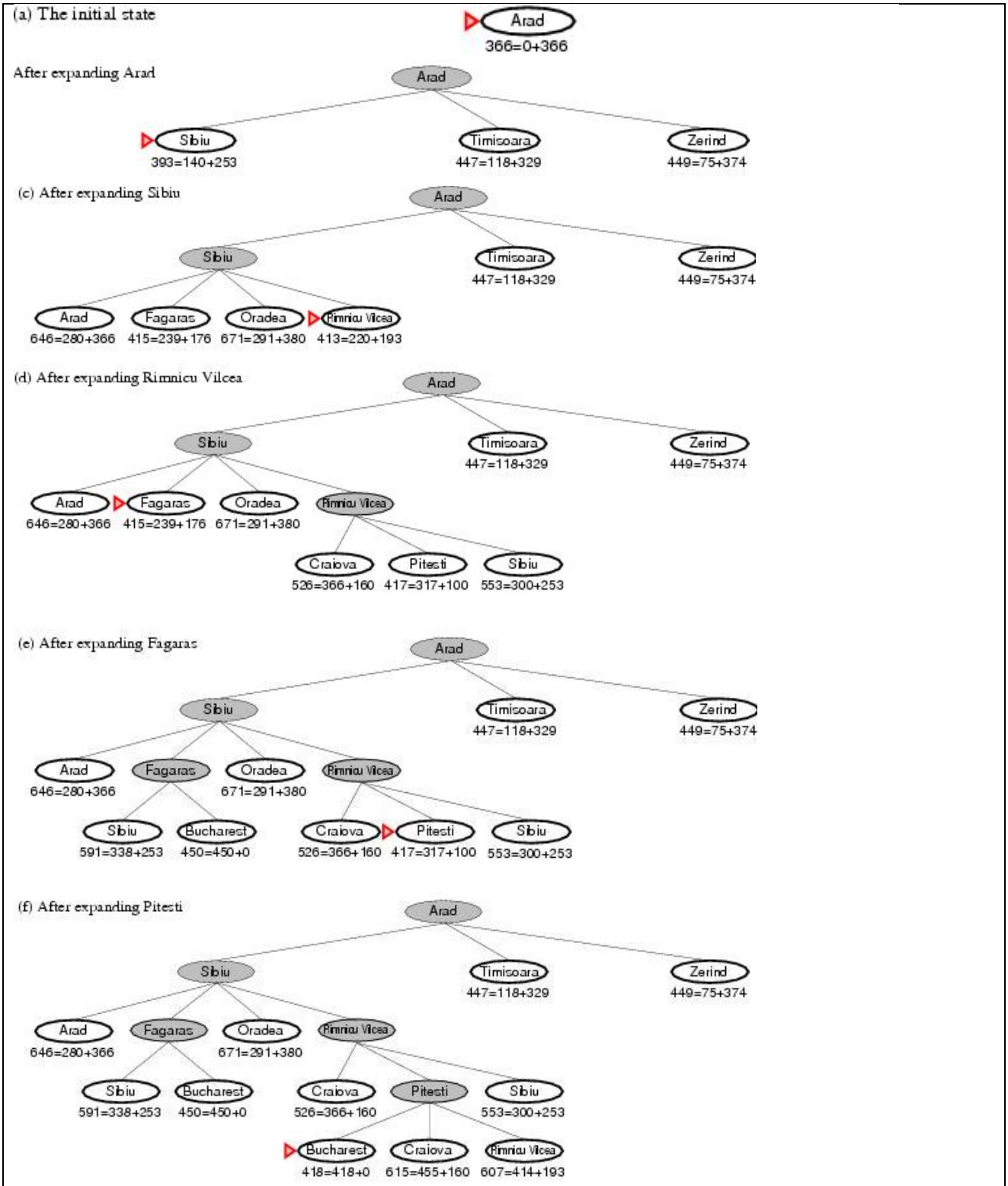


Figure 2.3 Stages in A* Search for Bucharest. Nodes are labeled with f = g + h . The h-values are the straight-line distances to Bucharest taken from figure 2.1

```
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

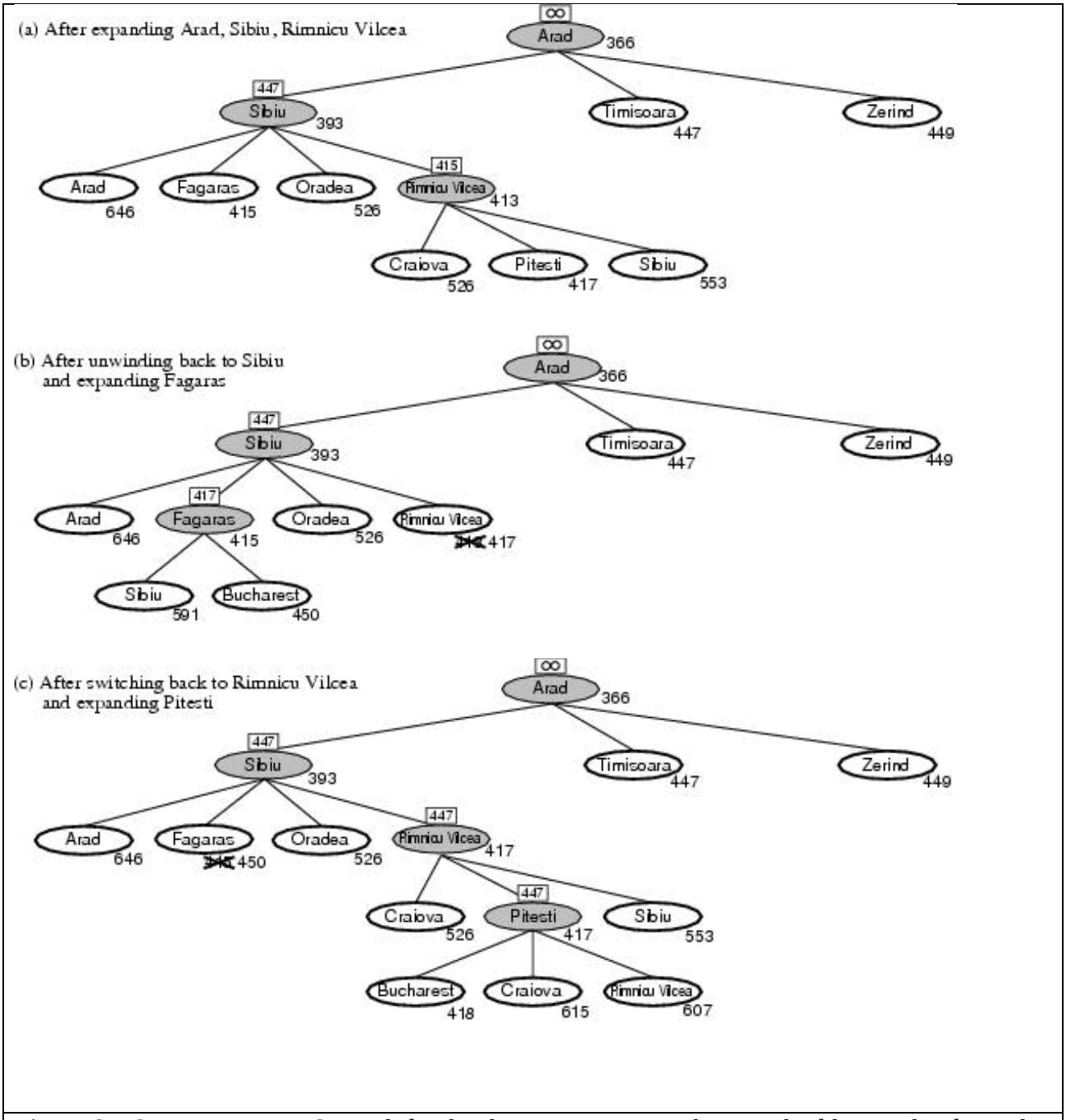


Figure 2.5 Stages in an RBFS search for the shortest route to Bucharest. The f-limit value for each recursive call is shown on top of each current node. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras).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

2.1.2 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

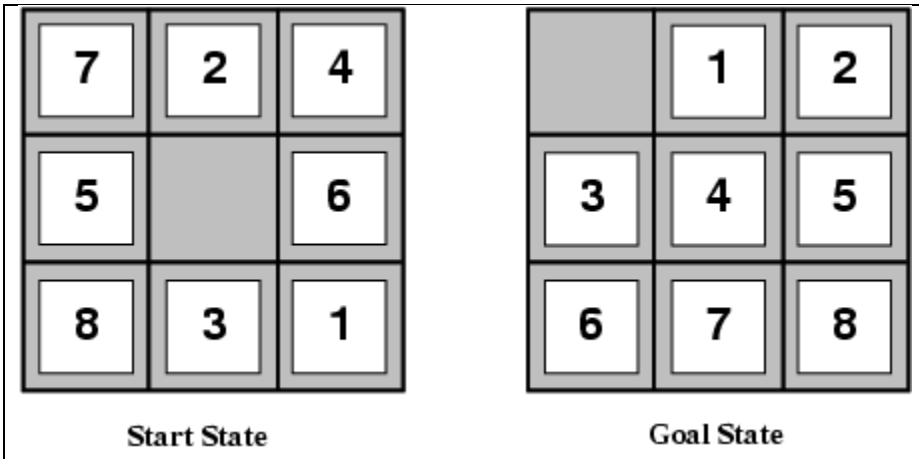


Figure 2.6 A typical instance of the 8-puzzle.
The solution is 26 steps long.

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) h_1 = the number of misplaced tiles.

For figure 2.6 ,all of the eight tiles are out of position,so the start state would have $h_1 = 8$. h_1 is an admissible heuristic.

(2) h_2 = the sum of the distances of the tiles from their goal positions. This is called **the city block distance** or **Manhattan distance.**

h_2 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

$h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18.$

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 h_1 and h_2,1200 random problems were generated with solution lengths from 2 to 24 and solved them with iterative deepening search and with A^* search using both h_1 and h_2. Figure 2.7 gives the averaghe number of nodes expanded by each strategy and the effective branching factor.

The results suggest that h_2 is better than h_1,and is far better than using iterative deepening search. For a solution length of 14,A^* with h_2 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 h_1,and h_2. 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

2.1.3 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**.

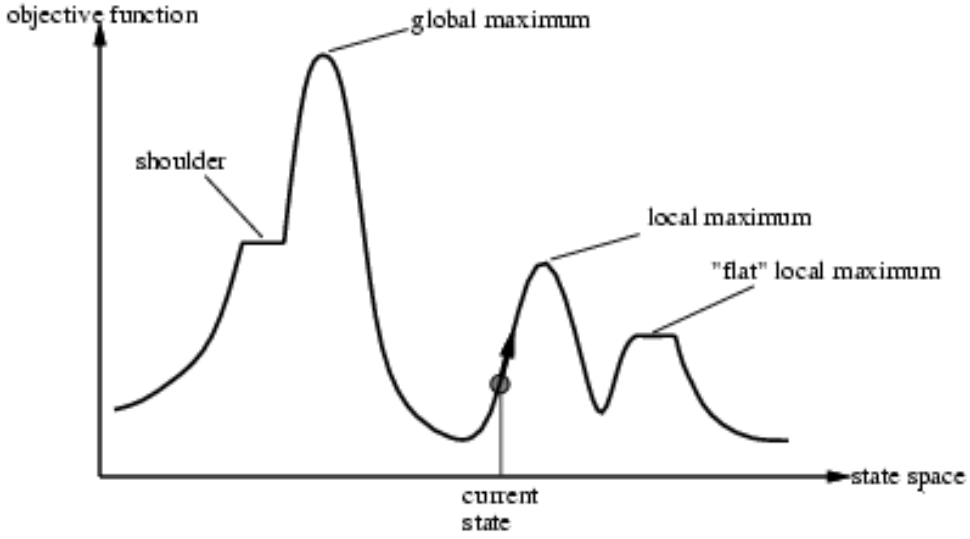


Figure 2.8 A one dimensional **state space landscape** in which elevation corresponds to the **objective function**. The aim is to find the global maximum. Hill climbing search modifies the current state to try to improve it ,as shown by the arrow. The various topographic features are defined in the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

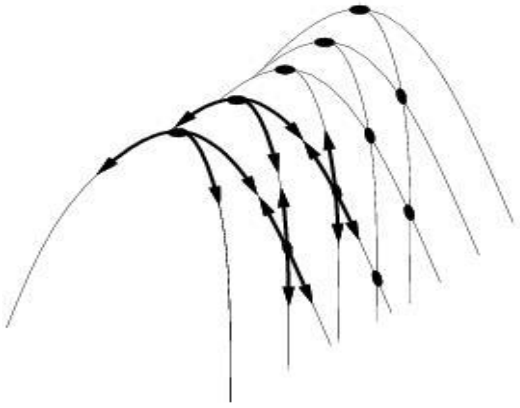


Figure 2.10 Illustration of why ridges cause difficulties for hill-climbing. The grid of states(dark circles) is superimposed on a ridge rising from left to right,creating a sequence of local maxima that are not directly connected to each other. From each local maximum,all th available options point 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 choosen 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 $\triangle$ 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 ∞ do
      T ← schedule[t]
      if T = 0 then return current
      next ← a randomly selected successor of current
      ΔE ← VALUE[next] – VALUE[current]
      if ΔE > 0 then current ← next
      else current ← next only with probability e^(Δ E/T)
```

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

Genetic algorithms

A Genetic algorithm(or GA) is a variant of stochastic beam search in which successor states are generated by combining two parent states,rather than by modifying a single state.
Like beam search,Gas begin with a set of k randomly generated states,called the population. Each state,or individual,is represented as a string over a finite alphabet – most commonly,a string of 0s and 1s. For example,an 8 8-quuens state must specify the positions of 8 queens,each in acolumn of 8 squares,and so requires 8 x $\log_2$ 8 = 24 bits.

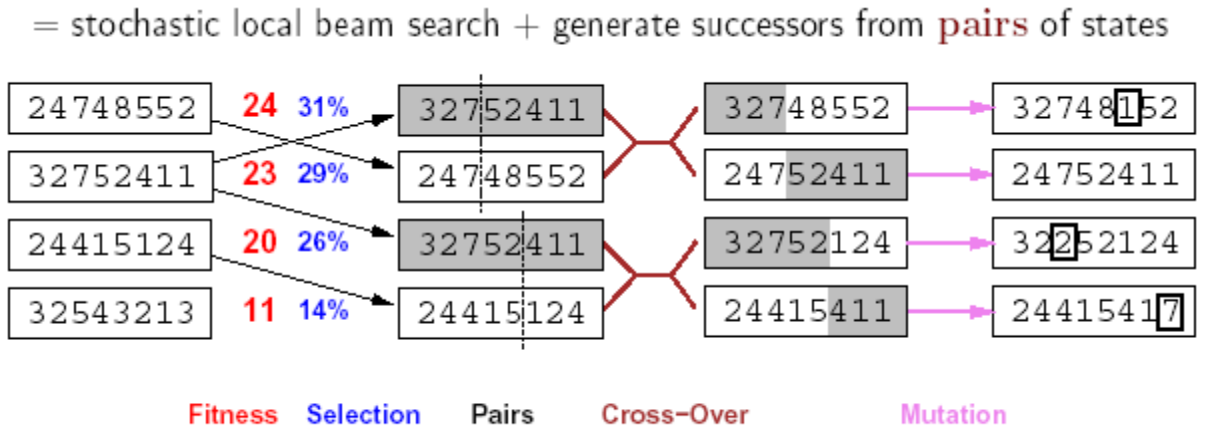


Figure 2.12 The genetic algorithm. The initial population in (a) is ranked by the fitness function in (b),resulting in pairs for mating in (c). They produce offspring in (d),which are subjected to

mutation in (e).

Figure 2.12 shows a population of four 8-digit strings representing 8-queen states. The production of the next generation of states is shown in Figure 2.12(b) to (e).
In (b) each state is rated by the evaluation function or the **fitness function**.
In (c),a random choice of two pairs is selected for reproduction,in accordance with the probabilities in (b).
Figure 2.13 describes the algorithm that implements all these steps.

```
function GENETIC_ALGORITHM( population, FITNESS-FN) return an individual
    input: population, a set of individuals
        FITNESS-FN, a function which determines the quality of the individual
    repeat
        new_population ← empty set
        loop for i from 1 to SIZE(population) do
            x ← RANDOM_SELECTION(population, FITNESS_FN)
    y ← RANDOM_SELECTION(population, FITNESS_FN)
            child ← REPRODUCE(x,y)
            if (small random probability) then child ← MUTATE(child )
            add child to new_population
        population ← new_population
    until some individual is fit enough or enough time has elapsed
    return the best individual
```

Figure 2.13 A genetic algorithm. The algorithm is same as the one diagrammed in Figure 2.12,with one variation:each mating of two parents produces only one offspring,not two.

2.1.4 LOCAL SEARCH IN CONTINUOUS SPACES

- We have considered algorithms that work only in discrete environments, but real-world environment are continuous
- Local search amounts to maximizing a continuous objective function in a multi-dimensional vector space.
- This is hard to do in general.
- Can immediately retreat
    - Discretize the space near each state
    - Apply a discrete local search strategy (e.g., stochastic hill climbing, simulated annealing)
- Often resists a closed-form solution
    - Fake up an empirical gradient
    - Amounts to greedy hill climbing in discretized state space
- Can employ Newton-Raphson Method to find maxima
- Continuous problems have similar problems: plateaus, ridges, local maxima, etc.

2.1.5 Online Search Agents and Unknown Environments

Online search problems

- Offline Search (all algorithms so far)

- Compute complete solution, ignoring environment Carry out action sequence

- Online Search
  - Interleave computation and action
  - Compute—Act—Observe—Compute—·
- Online search good
  - For dynamic, semi-dynamic, stochastic domains
  - Whenever offline search would yield exponentially many contingencies
- Online search necessary for exploration problem
  - States and actions unknown to agent
  - Agent uses actions as experiments to determine what to do

Examples

Robot exploring unknown building
Classical hero escaping a labyrinth

- Assume agent knows
  - Actions available in state s
    Step-cost function c(s,a,s′)
    State s is a goal state
- When it has visited a state s previously Admissible heuristic function h(s)
- Note that agent doesn't know outcome state (s′) for a given action (a) until it tries the action (and all actions from a state s)
- Competitive ratio compares actual cost with cost agent would follow if it knew the search space
- No agent can avoid dead ends in all state spaces
  - Robotics examples: Staircase, ramp, cliff, terrain
- Assume state space is safely explorable—some goal state is always reachable

Online Search Agents

- Interleaving planning and acting hamstrings offline search
  - A* expands arbitrary nodes without waiting for outcome of action Online algorithm can expand only the node it physically occupies Best to explore nodes in physically local order
  - Suggests using depth-first search
  - Next node always a child of the current
- When all actions have been tried, can't just drop state
  Agent must physically backtrack
- Online Depth-First Search
  - May have arbitrarily bad competitive ratio (wandering past goal) Okay for exploration; bad for minimizing path cost
- Online Iterative-Deepening Search
  - Competitive ratio stays small for state space a uniform tree

Online Local Search

- Hill Climbing Search
  - Also has physical locality in node expansions
    Is, in fact, already an online search algorithm
  - Local maxima problematic: can't randomly transport agent to new state in

effort to escape local maximum

- Random Walk as alternative
  - Select action at random from current state
  - Will eventually find a goal node in a finite space
  - Can be very slow, esp. if "backward" steps as common as "forward"
- Hill Climbing with Memory instead of randomness
  - Store "current best estimate" of cost to goal at each visited state Starting estimate is just h(s)
  - Augment estimate based on experience in the state space Tends to "flatten out" local minima, allowing progress Employ optimism under uncertainty
  - Untried actions assumed to have least-possible cost Encourage exploration of untried paths

Learning in Online Search

- Rampant ignorance a ripe opportunity for learning Agent learns a "map" of the environment
- Outcome of each action in each state
- Local search agents improve evaluation function accuracy
- Update estimate of value at each visited state
- Would like to infer higher-level domain model
- Example: "Up" in maze search increases y -coordinate Requires
- Formal way to represent and manipulate such general rules (so far, have hidden rules within the successor function)
- Algorithms that can construct general rules based on observations of the effect of actions

2.2 CONSTRAINT SATISFACTION PROBLEMS(CSP)

A **Constraint Satisfaction Problem**(or CSP) is defined by a set of **variables ,** $X_1, X_2, \ldots X_n$, and a set of constraints $C_1, C_2, \ldots, C_m$. Each variable X_i has a nonempty **domain** D,of possible **values**. Each constraint C_i involves some subset of variables and specifies the allowable combinations of values for that subset.

A **State** of the problem is defined by an **assignment** of values to some or all of the variables, $\{X_i = v_i, X_j = v_j, \ldots\}$. An assignment that does not violate any constraints is called a **consistent** or **legal assignment.** A complete assignment is one in which every variable is mentioned,and a **solution** to a CSP is a complete assignment that satisfies all the constraints.

Some CSPs also require a solution that maximizes an **objective function**.

Example for Constraint Satisfaction Problem :

Figure 2.15 shows the map of Australia showing each of its states and territories. We are given the task of coloring each region either red,green,or blue in such a way that the neighboring regions have the same color. To formulate this as CSP ,we define the variable to be the regions :WA,NT,Q,NSW,V,SA, and T. The domain of each variable is the set {red,green,blue}.The constraints require neighboring regions to have distinct colors;for example,the allowable combinations for WA and NT are the pairs {(red,green),(red,blue),(green,red),(green,blue),(blue,red),(blue,green)}.

The constraint can also be represented more succinctly as the inequality WA not = NT,provided the constraint satisfaction algorithm has some way to evaluate such expressions.) There are many possible solutions such as

{ WA = red, NT = green,Q = red, NSW = green, V = red ,SA = blue,T = red}.

It is helpful to visualize a CSP as a constraint graph,as shown in Figure 2.15(b). The nodes of the graph corresponds to variables of the problem and the arcs correspond to constraints.

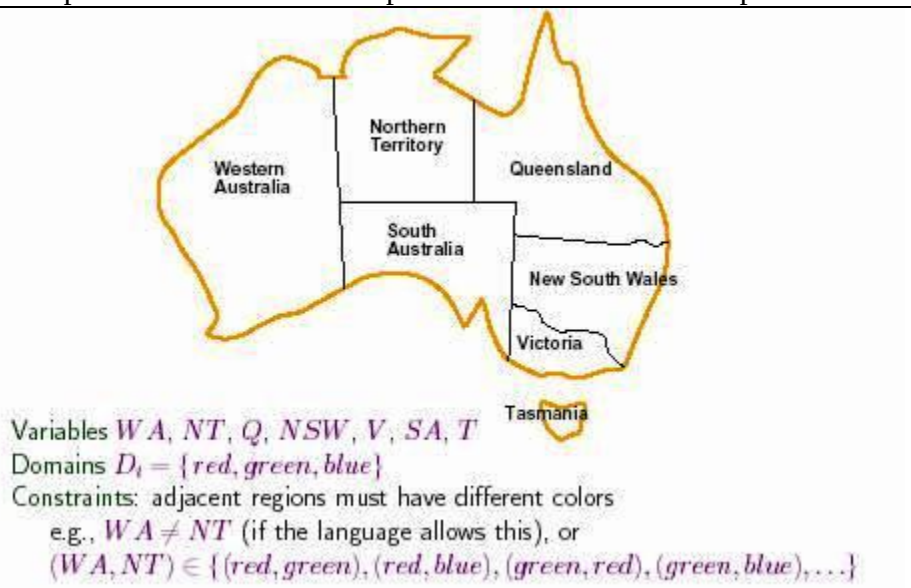


Figure 2.15 (a) Principle states and territories of Australia. Coloring this map can be viewed as aconstraint satisfaction problem. The goal is to assign colors to each region so that no neighboring regions have the same color.

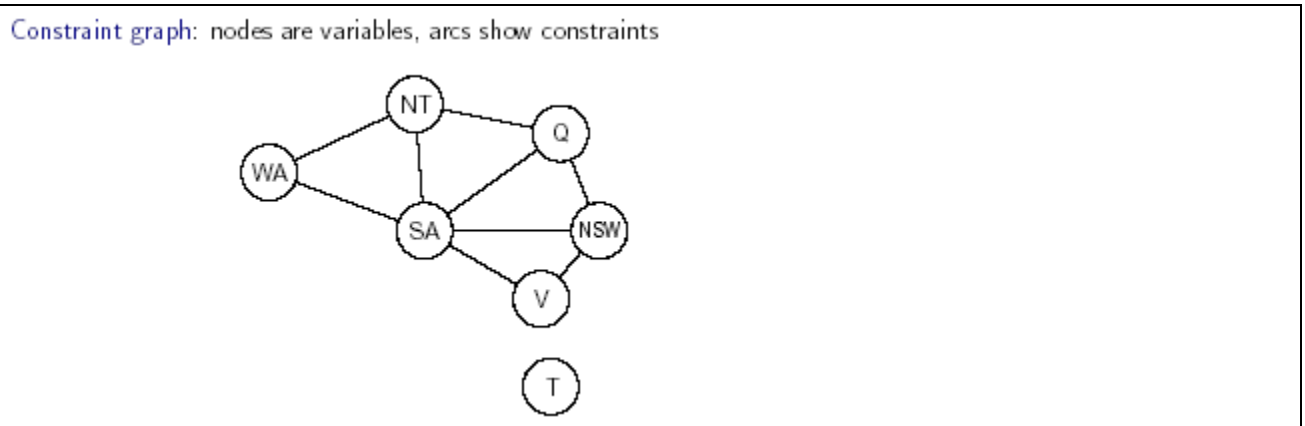


Figure 2.15 (b) The map coloring problem represented as a constraint graph.

CSP can be viewed as a standard search problem as follows :

- **Initial state** : the empty assignment {},in which all variables are unassigned.
- **Successor function** : a value can be assigned to any unassigned variable,provided that it does not conflict with previously assigned variables.
- **Goal test** : the current assignment is complete.
- **Path cost** : a constant cost(E.g.,1) for every step.

Every solution must be a complete assignment and therefore appears at depth n if there are n variables.

Depth first search algorithms are popular for CSPs

Varieties of CSPs

(i) Discrete variables

Finite domains

The simplest kind of CSP involves variables that are **discrete** and have **finite domains.** Map coloring problems are of this kind. The 8-queens problem can also be viewed as finite-domain

CSP,where the variables $Q_1,Q_2,\ldots..Q_8$ are the positions each queen in columns 1,….8 and each variable has the domain {1,2,3,4,5,6,7,8}. If the maximum domain size of any variable in a CSP is d,then the number of possible complete assignments is $O(d^n)$ – that is,exponential in the number of variables. Finite domain CSPs include **Boolean CSPs**,whose variables can be either *true* or *false*.

Infinite domains

Discrete variables can also have **infinite domains** – for example,the set of integers or the set of strings. With infinite domains,it is no longer possible to describe constraints by enumerating all allowed combination of values. Instead a constraint language of algebric inequalities such as $Startjob_1 + 5 <= Startjob_3$.

(ii) CSPs with continuous domains

CSPs with continuous domains are very common in real world. For example ,in operation research field,the scheduling of experiments on the Hubble Telescope requires very precise timing of observations; the start and finish of each observation and maneuver are continuous-valued variables that must obey a variety of astronomical,precedence and power constraints. The best known category of continuous-domain CSPs is that of **linear programming** problems,where the constraints must be linear inequalities forming a *convex* region. Linear programming problems can be solved in time polynomial in the number of variables.

Varieties of constraints :

(i) unary constraints involve a single variable.

Example : SA # green

(ii) Binary constraints involve paris of variables.

Example : SA # WA

(iii) Higher order constraints involve 3 or more variables.

Example : cryptarithmetic puzzles.

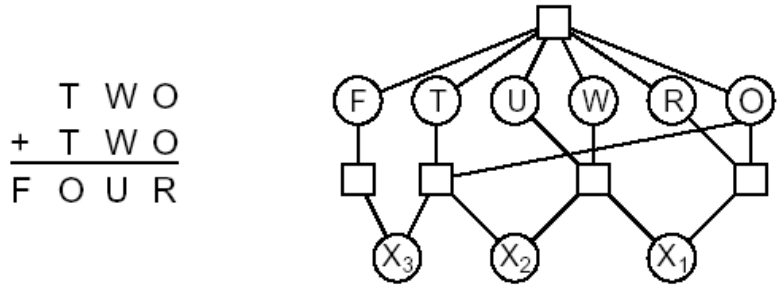


Figure 2.16 (a) Cryptarithmetic problem. Each letter stands for a distinct digit;the aim is to find a substitution of digits for letters such that the resulting sum is arithmetically correct,with the added restriction that no leading zeros are allowed. (b) The constraint hypergraph for the cryptarithmetic problem,showint the *Alldiff* constraint as well as the column addition constraints. Each constraint is a square box connected to the variables it contains.

2.2.2 Backtracking Search for CSPs

The term **backtracking search** is used for depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign. The algorithm is shown in figure 2.17.

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
  return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
      if value is consistent with assignment given CONSTRAINTS[csp] then
          add {var = value} to assignment
          result ← RECURSIVE-BACKTRACKING(assignment, csp)
          if result ≠ failure then return result
          remove {var = value} from assignment
  return failure
```

Figure 2.17 A simple backtracking algorithm for constraint satisfaction problem. The algorithm is modeled on the recursive depth-first search

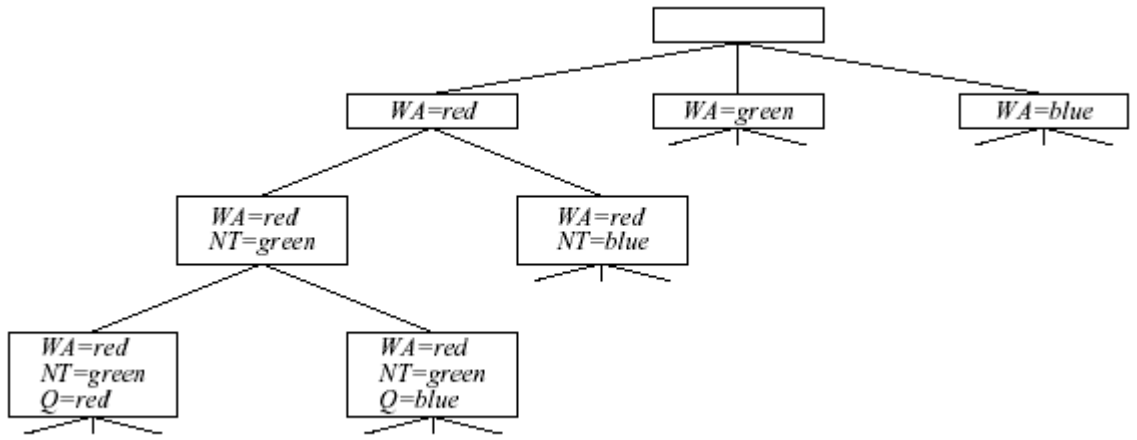


Figure 2.17(b) Part of search tree generated by simple backtracking for the map coloring problem.

Propagating information through constraints

So far our search algorithm considers the constraints on a variable only at the time that the variable is chosen by SELECT-UNASSIGNED-VARIABLE. But by looking at some of the constraints earlier in the search, or even before the search has started, we can drastically reduce the search space.

Forward checking

One way to make better use of constraints during search is called **forward checking**. Whenever a variable X is assigned, the forward checking process looks at each unassigned variable Y that is connected to X by a constraint and deletes from Y 's domain any value that is inconsistent with the value chosen for X. Figure 5.6 shows the progress of a map-coloring search with forward checking.

	WA	NT	Q	NSW	V	SA	T
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After WA=red	(R)	G B	R G B	R G B	R G B	G B	R G B
After Q=green	(R)	B	(G)	R B	R G B	B	R G B
After V=blue	(R)	B	(G)	R	(B)		R G B

Figure 5.6 The progress of a map-coloring search with forward checking. $WA = red$ is assigned first; then forward checking deletes red from the domains of the neighboring variables NT and SA. After $Q = green$, $green$ is deleted from the domains of NT, SA, and NSW. After $V = blue$, $blue$ is deleted from the domains of NSW and SA, leaving SA with no legal values.

Constraint propagation

Although forward checking detects many inconsistencies, it does not detect all of them.

Constraint propagation is the general term for propagating the implications of a constraint on one variable onto other variables.

Arc Consistency

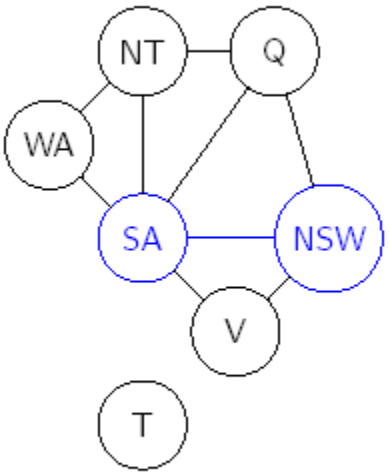


Figure: Australian Territories

- One method of constraint propagation is to enforce **arc consistency**
  - Stronger than forward checking
  - Fast
- *Arc* refers to a *directed* arc in the constraint graph
- Consider two nodes in the constraint graph (e.g., *SA* and *NSW*)
  - An arc is **consistent** if
  - For every value x of *SA*
  - There is some value y of *NSW* that is consistent with x
- Examine arcs for consistency in *both* directions

k-Consistency

- Can define stronger forms of consistency

k-Consistency

A CSP is *k*-**consistent** if, for any consistent assignment to $k-1$ variables, there is a consistent assignment for the *k*-th variable

- **1-consistency (node consistency)**
  - Each variable by itself is consistent (has a non-empty domain)
- **2-consistency (arc consistency)**
- **3-consistency (path consistency)**
  - Any pair of adjacent variables can be extended to a third

Local Search for CSPs

- Local search algorithms good for many CSPs
- Use complete-state formulation
  - Value assigned to every variable
  - Successor function changes one value at a time
- Have already seen this
  - Hill climbing for 8-queens problem (AIMA § 4.3)
- Choose values using **min-conflicts** heuristic
  - Value that results in the minimum number of conflicts with other variables

2.2.3 The Structure of Problems
Problem Structure

- Consider ways in which the structure of the problem's constraint graph can help find solutions
- Real-world problems require decomposition into subproblems

Independent Subproblems

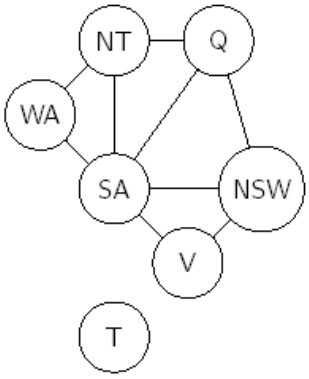


Figure: Australian Territories

- T is not connected
- Coloring T and coloring remaining nodes are **independent subproblems**
- *Any* solution for T combined with *any* solution for remaining nodes solves the problem
- Independent subproblems correspond to **connected components** of the constraint graph
- Sadly, such problems are rare

Tree-Structured CSPs

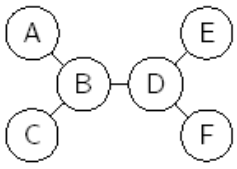


Figure: Tree-Structured CSP

- In most cases, CSPs are connected
- A simple case is when the constraint graph is a **tree**
- Can be solved in time linear in the number of variables
  - Order variables so that each parent precedes its children
  - Working "backward," apply arc consistency between child and parent
  - Working "forward," assign values consistent with parent

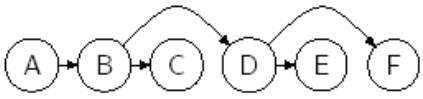


Figure: Linear ordering

2.4 ADVERSARIAL SEARCH

Competetive environments,in which the agent's goals are in conflict,give rise to **adversarial search** problems – often known as **games.**

2.4.1 Games

Mathematical **Game Theory**,a branch of economics,views any **multiagent environment** as a **game** provided that the impact of each agent on the other is "significant",regardless of whether the agents are cooperative or competitive. In,**AI**,"games" are deterministic,turn-taking,two-player,zero-sum games of perfect information. This means deterministic,fully observable environments in which there are two agents whose actions must alternate and in which the **utility values** at the end of the game are always equal and opposite. For example,if one player wins the game of chess(+1),the other player necessarily loses(-1). It is this opposition between the agents' utility functions that makes the situation **adversarial.**

Formal Definition of Game

We will consider games with two players,whom we will call **MAX** and **MIN**. MAX moves first,and then they take turns moving until the game is over. At the end of the game, points are awarded to the winning player and penalties are given to the loser. A **game** can be formally defined as a **search problem** with the following components :

- The **initial state**,which includes the board position and identifies the player to move.
- A **successor function**,which returns a list of (*move,state*) pairs,each indicating a legal move and the resulting state.
- A **terminal test**,which describes when the game is over. States where the game has ended are called **terminal states**.
- A **utility function** (also called an objective function or payoff function),which give a numeric value for the terminal states. In chess,the outcome is a win,loss,or draw,with values +1,-1,or 0. he payoffs in backgammon range from +192 to -192.

Game Tree

The **initial state** and **legal moves** for each side define the **game tree** for the game. Figure 2.18 shows the part of the game tree for tic-tac-toe (noughts and crosses). From the initial state,MAX has nine possible moves. Play alternates between MAX's placing an X and MIN's placing a 0 until we reach leaf nodes corresponding to the terminal states such that one player has three in a row or all the squares are filled. He number on each leaf node indicates the utility value of the terminal state from the point of view of MAX;high values are assumed to be good for MAX and bad for MIN. It is the MAX's job to use the search tree(particularly the utility of terminal states) to determine the best move.

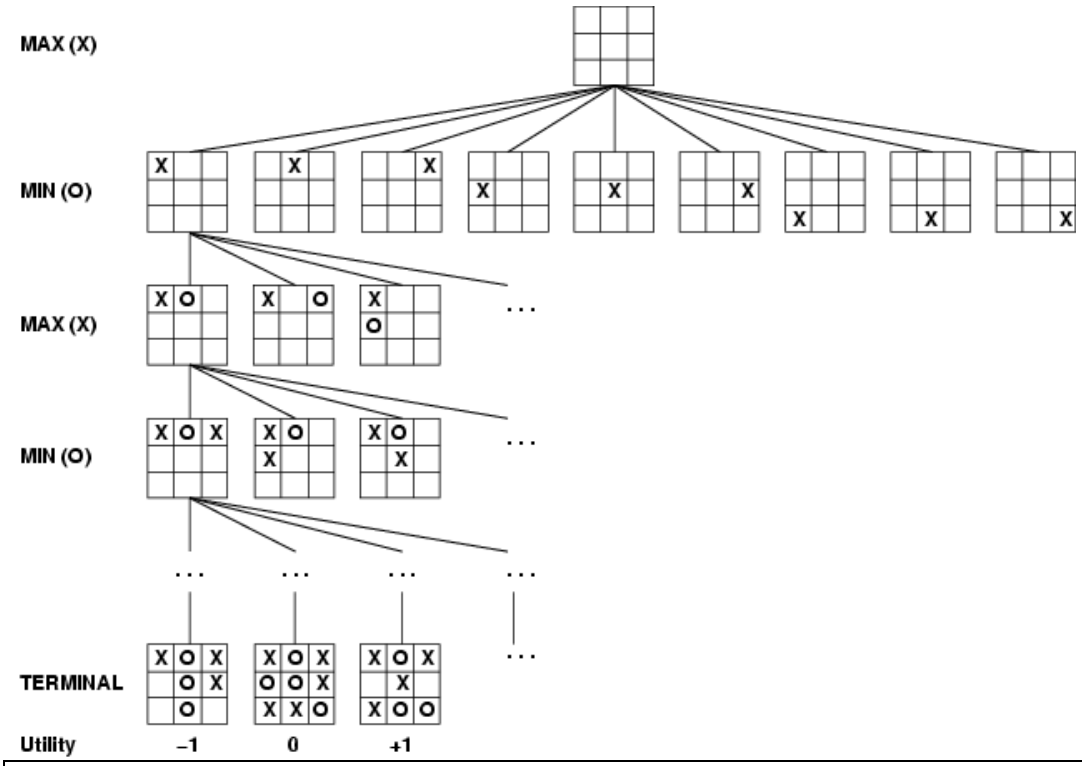


Figure 2.18 A partial search tree . The top node is the initial state,and MAX move first,placing an X in an empty square.

2.4.2 Optimal Decisions in Games

In normal search problem,the **optimal solution** would be a sequence of move leading to a **goal state** – a terminal state that is a win. In a game,on the other hand,MIN has something to say about it,MAX therefore must find a contingent **strategy**,which specifies MAX's move in the **initial state**,then MAX's moves in the states resulting from every possible response by MIN,then MAX's moves in the states resulting from every possible response by MIN those moves,and so on. An **optimal strategy** leads to outcomes at least as good as any other strategy when one is playing an infallible opponent.

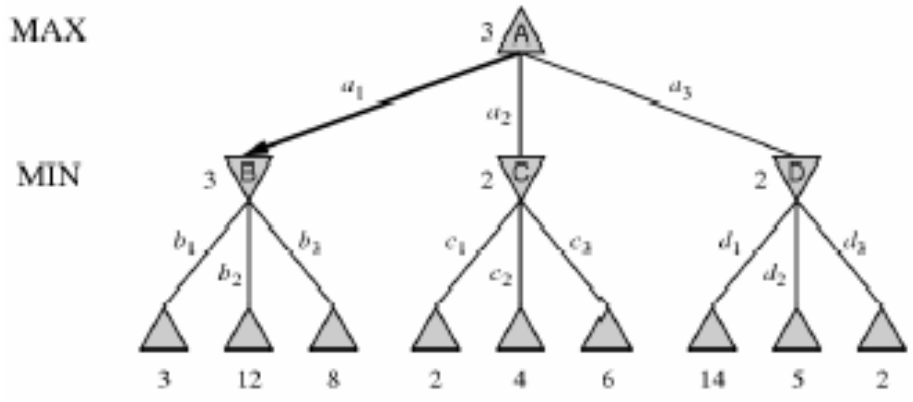


Figure 2.19 A two-ply game tree. The △ nodes are "MAX nodes",in which it is AMX's turn to move,and the ▽ nodes are "MIN nodes". The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1,because it leads to the successor with the highest minimax value,and MIN's best reply is b_1,because it leads to the successor with the lowest minimax value.

Minimax Search: Algorithm

```
function MINIMAX-DECISION(state) returns an action
  inputs: state, current state in game

  v ← MAX-VALUE(state)
  return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value          For MAX Node
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for a, s in SUCCESSORS(state) do
    v ← MAX(v, MIN-VALUE(s))
  return v

function MIN-VALUE(state) returns a utility value          For MIN Node
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a, s in SUCCESSORS(state) do
    v ← MIN(v, MAX-VALUE(s))
  return v
```

Figure 2.20 An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move,that is,the move that leads to the outcome with the best utility,under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree,all the way to the leaves,to determine the backed-up value of a state.

The minimax Algorithm

The minimax algorithm(Figure 2.20) computes the minimax decision from the current state. It uses a simple recursive computation of the minimax values of each successor state,directly implementing the defining equations. The recursion proceeds all the way down to the leaves of the tree ,and then the minimax values are **backed up** through the tree as the recursion unwinds**.** For example in Figure 2.19,the algorithm first recourses down to the three bottom left nodes,and uses the utitliy function on them to discover that their values are 3,12,and 8 respectively. Then it takes the minimum of these values,3,and returns it as the backed-up value of node B. A similar process gives the backed up values of 2 for C and 2 for D. Finally,we take the maximum of 3,2,and 2 to get the backed-up value of 3 at the root node. The minimax algorithm performs a complete depth-first exploration of the game tree. If the maximum depth of the tree is *m*,and there are b legal moves at each point,then the time

complexity of the minimax algorithm is $O(b^m)$. The space complexity is O(bm) for an algorithm that generates successors at once.

2.4.3 Alpha-Beta Pruning

The problem with minimax search is that the number of game states it has to examine is **exponential** in the number of moves. Unfortunately,we can't eliminate the exponent,but we can effectively cut it in half. By performing **pruning,**we can eliminate large part of the tree from consideration. We can apply the technique known as **alpha beta pruning** ,when applied to a minimax tree ,it returns the same move as **minimax** would,but **prunes away** branches that cannot possibly influence the final decision.

Alpha Beta pruning gets its name from the following two parameters that describe bounds on the backed-up values that appear anywhere along the path:

- α : the value of the best(i.e.,highest-value) choice we have found so far at any choice point along the path of MAX.
- β: the value of best (i.e., lowest-value) choice we have found so far at any choice point along the path of MIN.

Alpha Beta search updates the values of α and β as it goes along and prunes the remaining branches at anode(i.e.,terminates the recursive call) as soon as the value of the current node is known to be worse than the current α and β value for MAX and MIN,respectively. The complete algorithm is given in Figure 2.21.

The effectiveness of alpha-beta pruning is highly dependent on the order in which the successors are examined. It might be worthwhile to try to examine first the successors that are likely to be the best. In such case,it turns out that alpha-beta needs to examine only $O(b^{d/2})$ nodes to pick the best move,instead of $O(b^d)$ for minimax. This means that the effective branching factor becomes sqrt(b) instead of b – for chess,6 instead of 35. Put anotherway alpha-beta cab look ahead roughly twice as far as minimax in the same amount of time.

```
function ALPHA-BETA-SEARCH(state) returns an action
   inputs: state, current state in game
   v ← MAX-VALUE(state, −∞, +∞)
   return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
   inputs: state, current state in game
           α, the value of the best alternative for MAX along the path to state
           β, the value of the best alternative for MIN along the path to state
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for a, s in SUCCESSORS(state) do
      v ← MAX(v, MIN-VALUE(s, α, β))
      if v ≥ β then return v
      α ← MAX(α, v)
   return v
```

```
function MIN-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for a, s in SUCCESSORS(state) do
    v ← MIN(v, MAX-VALUE(s, α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Figure 2.21 The alpha beta search algorithm. These routines are the same as the minimax routines in figure 2.20,except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β

2.4.4 Imperfect ,Real-time Decisions

The minimax algorithm generates the entire game search space,whereas the alpha-beta algorithm allows us to prune large parts of it. However,alpha-beta still has to search all the way to terminal states for atleast a portion of search space. Shannon's 1950 paper,Programming a computer for playing chess,proposed that programs should **cut off** the search earlier and apply a heuristic **evaluation function** to states in the search,effectively turning nonterminal nodes into terminal leaves. The basic idea is to alter minimax or alpha-beta in two ways :

(1) The utility function is replaced by a heuristic evaluation function EVAL,which gives an estimate of the position's utility,and

(2) the terminal test is replaced by a **cutoff test** that decides when to apply EVAL.

2.4.5 Games that include Element of Chance

Evaluation functions

An evaluation function returns an estimate of the expected utility of the game from a given position,just as the heuristic function return an estimate of the distance to the goal.

Games of imperfect information

- Minimax and alpha-beta pruning require too much leaf-node evaluations.
  May be impractical within a reasonable amount of time.
- SHANNON (1950):
  - Cut off search earlier (replace TERMINAL-TEST by CUTOFF-TEST)
  - Apply heuristic evaluation function EVAL (replacing utility function of alpha-beta)

Cutting off search

Change**:**

- **if** TERMINAL-TEST(*state*) **then return** UTILITY(*state*)

  into
- **if** CUTOFF-TEST(*state,depth*) **then return** EVAL(*state*)

Introduces a fixed-depth limit *depth*

- Is selected so that the amount of time will not exceed what the rules of the game allow.

When cuttoff occurs, the evaluation is performed.

Heuristic EVAL

Idea: produce an estimate of the expected utility of the game from a given position.

Performance depends on quality of EVAL.

Requirements:

- EVAL should order terminal-nodes in the same way as UTILITY.
- Computation may not take too long.
- For non-terminal states the EVAL should be strongly correlated with the actual chance of winning.

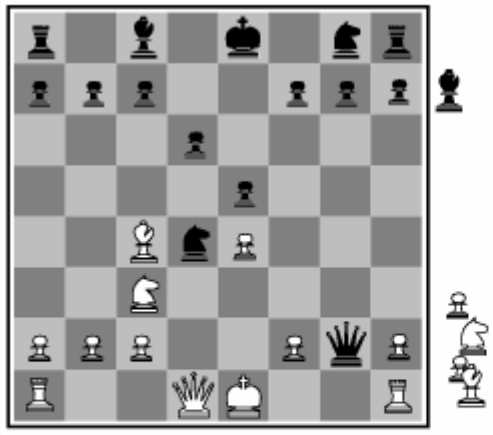
Only useful for quiescent (no wild swings in value in near future) states

Weighted Linear Function

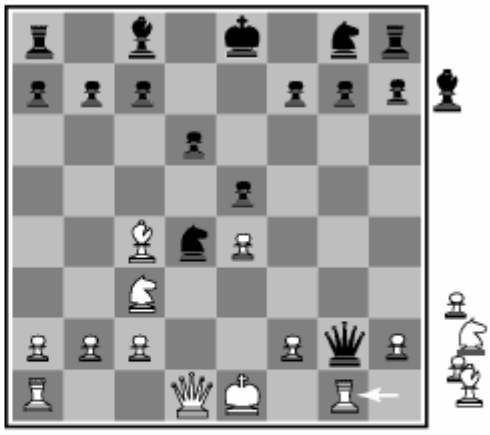
The introductory chess books give an approximate material value for each piece : each pawn is worth 1,a knight or bishop is worth 3,a rook 3,and the queen 9. These feature values are then added up toobtain the evaluation of the position. Mathematically,these kind of evaluation fuction is called weighted linear function,and it can be expressed as :

$$Eval(s) = w1\ f1(s) + w2\ f2(s) + \ldots + wn\ fn(s)$$

- e.g., w1 = 9 with
  f1(s) = (number of white queens) – (number of black queens), etc.



(a) White to move



(b) White to move

(a) Black has an advantage of a knight and two pawns and will win the game.
(b) Black will lose after white captures the queen.

Games that include chance

In real life,there are many unpredictable external events that put us into unforeseen situations. Many games mirror this unpredictability by including a random element,such as throwing a dice. **Backgammon** is a typical game that combines luck and skill. Dice are rolled at the beginning of player's turn to determine the legal moves. The backgammon position of Figure 2.23,for example,white has rolled a 6-5,and has four possible moves.

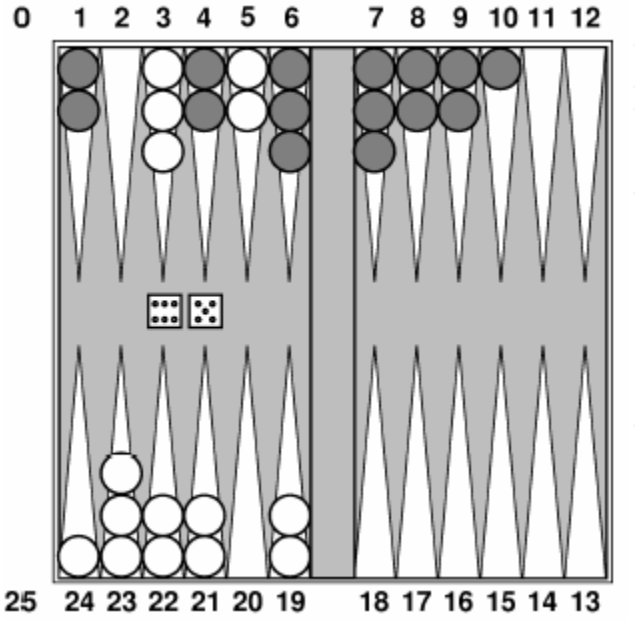


Figure 2.23 A typical backgammon position. The goal of the game is to move all one's pieces off the board. White moves clockwise toward 25,and black moves counterclockwise toward 0. A piece can move to any position unless there are multiple opponent pieces there; if there is one opponent ,it is captured and must start over. In the position shown,white has rolled 6-5 and must choose among four legal moves (5-10,5-11),(5-11,19-24),(5-10,10-16),and (5-11,11-16)

- White moves clockwise toward 25
- Black moves counterclockwise toward 0
- A piece can move to any position unless there are multiple opponent pieces there; if there is one opponent, it is captured and must start over.
- White has rolled 6-5 and must choose among four legal moves: (5-10, 5-11), (5-11, 19-24) (5-10, 10-16), and (5-11, 11-16)

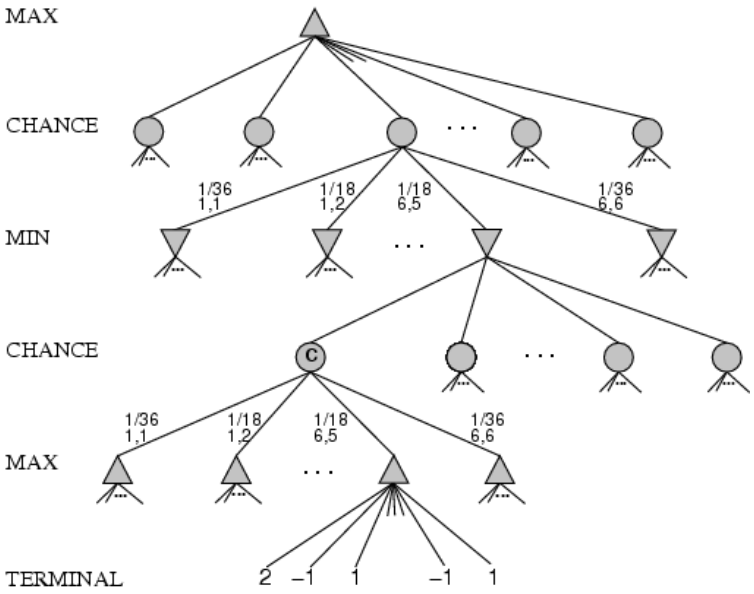


Figure 2-24 Schematic game tree for a backgammon position.

Expected minimax value

EXPECTED-MINIMAX-VALUE(*n*)=

UTILITY(*n*) If *n* is a terminal

MINIMAX-VALUE(*s*) max*s* ∈ *successors(n)* If *n* is a max node

MINIMAX-VALUE(*s*) min*s* ∈ *successors(n)* If *n* is a max node

EXPECTEDMINIMAX(*s*) $\sum s \in successors(n)\ P(s)$. If *n* is a chance node

These equations can be backed-up recursively all the way to the root of the game tree.
