SCSX1021 - ARTIFICIAL INTELLIGENCE

UNIT – II

Informed Search

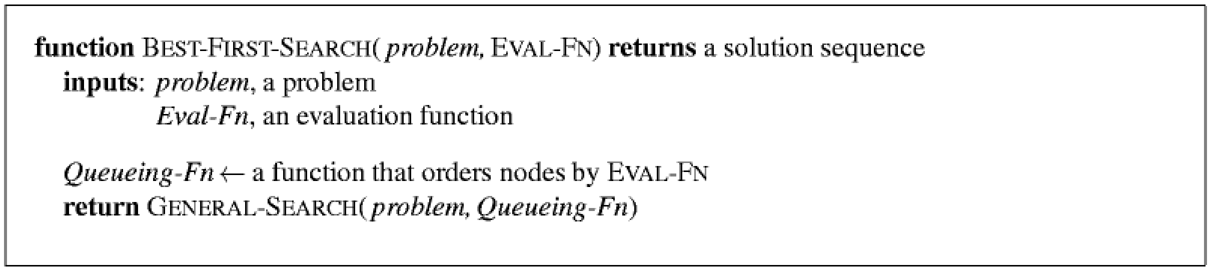
Introduction

- Informed search strategies use problem specific knowledge beyond the definition of the problem itself.
- Use the knowledge of the problem domain to build an evaluation function f.
- For every node n is the search space $f(n)$ quantifiers the desirability of expanding n in order to reach the goal.
- To solve large problem with large number of possible states problem specific knowledge need to be added to increase the efficiency of search algorithm.
- Can find solution more efficiently than a uniformed strategy.
- A key point of informed search strategy is heuristic function. So it is called as **heuristic function.**

Strategies

Best First Search:

- Best First Search is an instance of the general TREE SEARCH or GRAPH SEARCH algorithm in which a node is selected for expansion based on an evaluation function$f(n)$.
- The Best First Search algorithms have different evaluation functions. A key component of these algorithms is a heuristic function denoted $h(n)$.
- $h(n)$ = estimated cost of the cheapest path from node n to a goal node.
- **For example:** From Figure 2.1**,** in Romania, one might estimate the cost of the cheapest path from Arad to Bucharest via the straight line distance from Arad to Bucharest.
- **Heuristic functions are** the most common form in which additional knowledge of the problem is imparted to the search algorithm.
- It can implemented as:

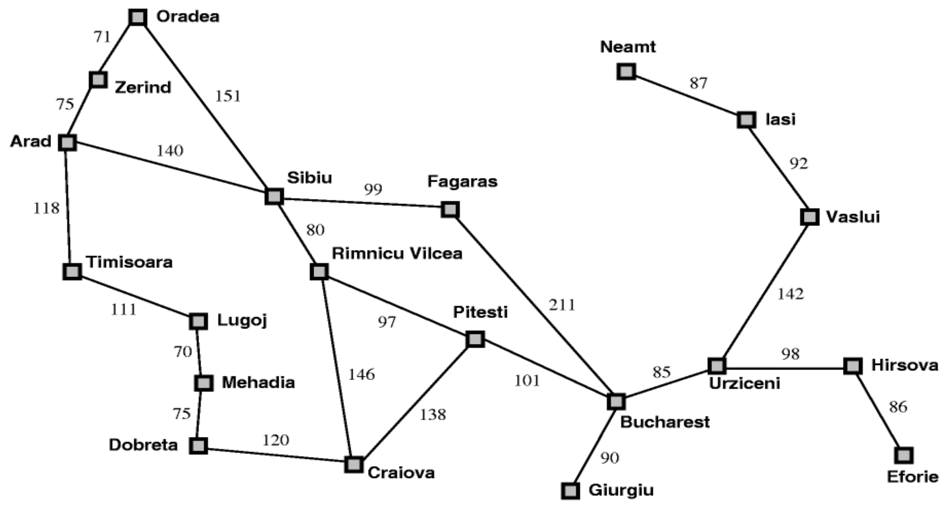


```
function BEST-FIRST-SEARCH( problem, EVAL-FN) returns a solution sequence
  inputs: problem, a problem
          Eval-Fn, an evaluation function

  Queueing-Fn ← a function that orders nodes by EVAL-FN
  return GENERAL-SEARCH( problem, Queueing-Fn)
```

Greedy Best First Search:

- ➢ The simplest best-first strategy is to minimize the estimated cost to reach the goal, i.e., always expand the node that appears to be closest to the goal. A function that calculates cost estimates is called an **heuristic function**.
- ➢ h (n) = estimated cost of the cheapest path from the state at n to a goal state
- ➢ A best-first search that uses h to select the next node to expand is called a **greedy search**.
- ➢ To get an idea of what an heuristic function looks like, lets look at a particular problem.
- ➢ Here we can use as h the **straight-line distance** to the goal.
- ➢ To do this, we need the map co-ordinates of each city.
- ➢ This heuristic works because roads tend to head in more or less of a straight line.



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 Road Map

This figure 2.2 shows the progress of a greedy search to find a path from Arad to

Bucharest.

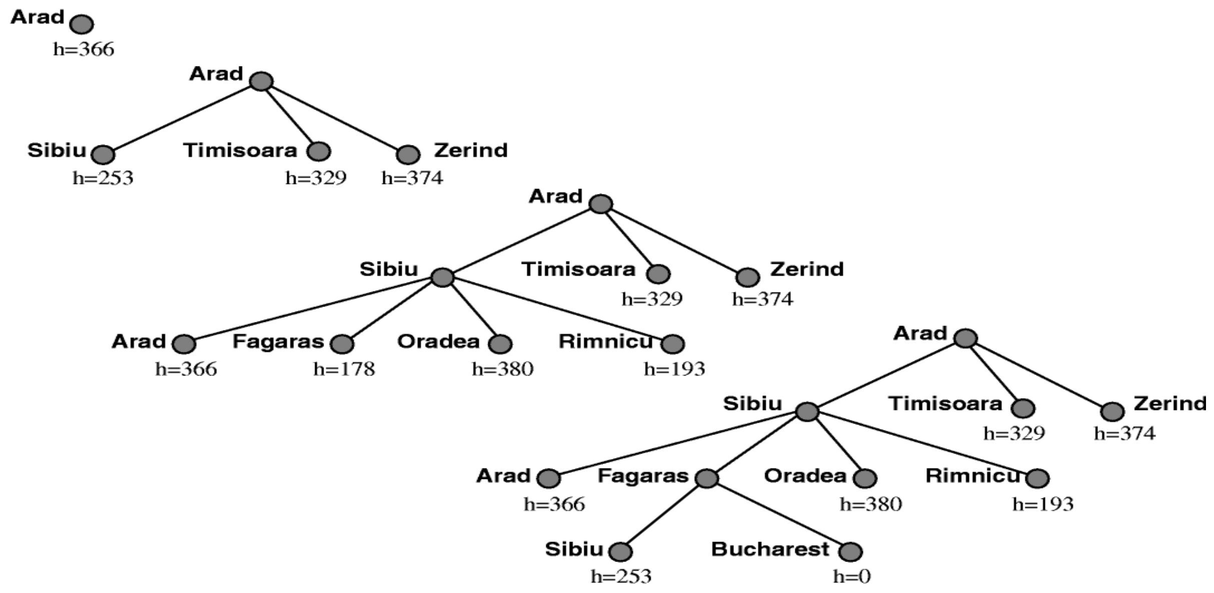


Figure 2.2 Search for Bucharest using the straight line distance heuristic

- For this problem, greedy search leads to a minimal cost search because no node off the solution path is expanded.
- However, it does not find the optimal path: the path it found via Sibiu and Fagaras to Bucharest is 32 miles longer than the path through Pimnicu Vilcea and Pitesti.
- Hence, the algorithm always chooses what looks locally best, rather than worrying about whether or not it will be best in the long run. (This is why its called greedy search.) Greedy search is susceptible to false starts.
- Consider the problem of getting from Iasi to Fagaras. h suggests that Neamt be expanded first, but it is a dead end.
- The solution is to go first to Vaslui and then continue to Urziceni, Bucharest and Fagaras. Note that if we are not careful to detect repeated states, the solution will never be found - the search will oscillate between Neamt and Iasi.
- Greedy search resembles dfs in the way that it prefers to follow a single path to the goal and backup only when a dead end is encountered.
- It suffers from the same defects as dfs -it is not optimal and it is incomplete because it can start down an infinite path and never try other possibilities.

- The worst-case complexity for greedy search is $O(b^m)$, where m is the maximum depth of the search.
- Its space complexity is the same as its time complexity, but the worst case can be substantially reduced with a good heuristic function.

A* Heuristic Function:(Minimizing the total estimated solution cost)

A* Search is the most widely used form of best-first search. **A* Search** evaluates the node by combining

g(n) = the cost to reach the node, and

h(n) = the cost to get from the node to the **goal** :

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

where

f(n) -> Cheapest solution cost.

g(n) -> path cost from the start node to the node "n"

h(n) -> cheapest path cost from the node "n" to the goal node

A* Search is both optimal and complete. A* is optimal if h(n) is an admissible heuristic. The obvious example of admissible heuristic is the straight-line distance hSLD.

It cannot be an overestimate.

A* Search is optimal if h(n) is an admissible heuristic – that is, provided that h(n) never overestimates the cost to reach the goal.

An obvious example of an admissible heuristic is the straight-line distance hSLD that we usedin getting to Bucharest. The progress of an A* tree search for Bucharest is shown in Figure 2.3.

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

An obvious example of an admissible heuristic is the straight-line distance *hSLD* that we used in getting to Bucharest. Straight-line distance is admissible because the shortest path between any two points is a straight line, so the straight line cannot be an overestimate.

In Figure 2.3, we show the progress of an A* tree search for Bucharest. The values of *g* are computed from the step costs in Figure 2.1, and the values of *hSLD* are given in Figure 2.3.

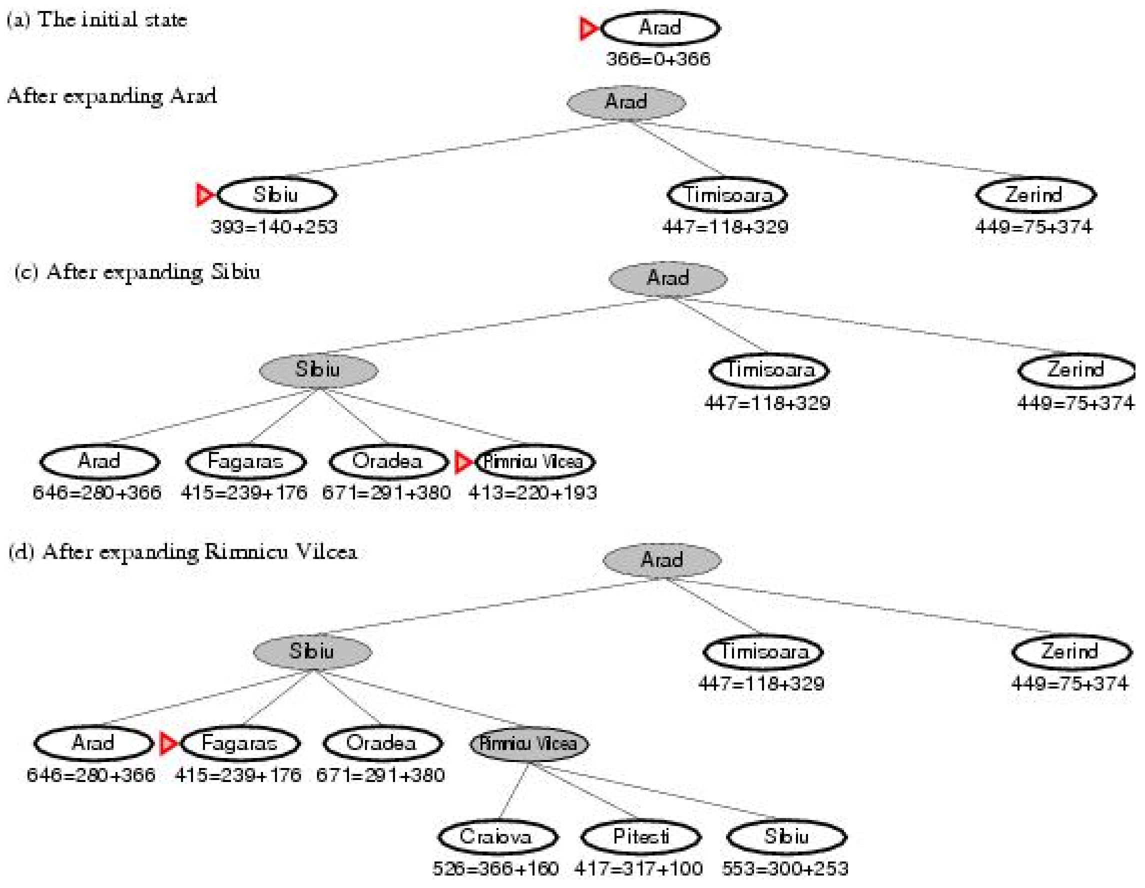
Notice in particular that Bucharest first appears on the fringe at step (e), but it is not selected for expansion because its *f-cost* (450) is higher than that of Pitesti (417). Another way to say this is that there *might* be a solution through Pitesti whose cost is as low as 417, so the algorithm will not settle for a solution that costs 450.

Optimality of A*

- Provided that h(n) never overestimates the cost to reach the goal, then in tree search A* gives the optimal solution.
- Suppose G2 is a suboptimal goal node generated to the tree.
- Let C* be the cost of the optimal solution .
- Because G2 is a goal node, it holds that h(G2) = 0, and we know that f(G2) = g(G2) > C*.
- On the other hand, if a solution exists, there must exist a node n that is on the optimal solution path in the tree.
- Because h(n) does not overestimate the cost of completing the solution path, f(n) = g(n) + h(n) C*.
- We have shown that f(n) C* < f(G2), so G2 will not be expanded and A* must return an optimal solution.
- In graph search finding an optimal solution requires taking care that the optimal solution is not discarded in repeated states .
- A particularly important special case are consistent (or monotonic) heuristics for which the triangle inequality holds in form h(n) c([n,a], n') + h(n'), where n'� S(n) (the chosen action is a) and c([n,a], n') is the step cost.
- Straight-line distance is also a monotonic heuristic.
- A* using a consistent heuristic h(n) is optimal also for graph search.
- If h(n) is consistent, the values of f(n) along any path are no decreasing.
- Suppose that n' is a successor of n so that g(n') = c([n,a], n') + g(n) f(n') = g(n') + h(n') = c([n,a], n') + g(n) + h(n') • g(n) + h(n) = f(n).
- Hence, the first goal node selected for expansion (in graph search) must be an optimal solution.
- In looking for a solution, A* expands all nodes n for which f(n) < C*, and some of those for which f(n) = C* .
- However, all nodes n for which f(n) > C* get pruned.
  It is clear that A* search is complete .
- A* search is also optimally efficient for any given heuristic function, because any algorithm that does not expand all nodes with f(n) < C* runs the risk of missing the optimal solution.
- Despite being complete, optimal, and optimally efficient, A* search also has its weaknesses.

- The number of nodes for which f(n) < C* for most problems is exponential in the length of the solution.

Example:



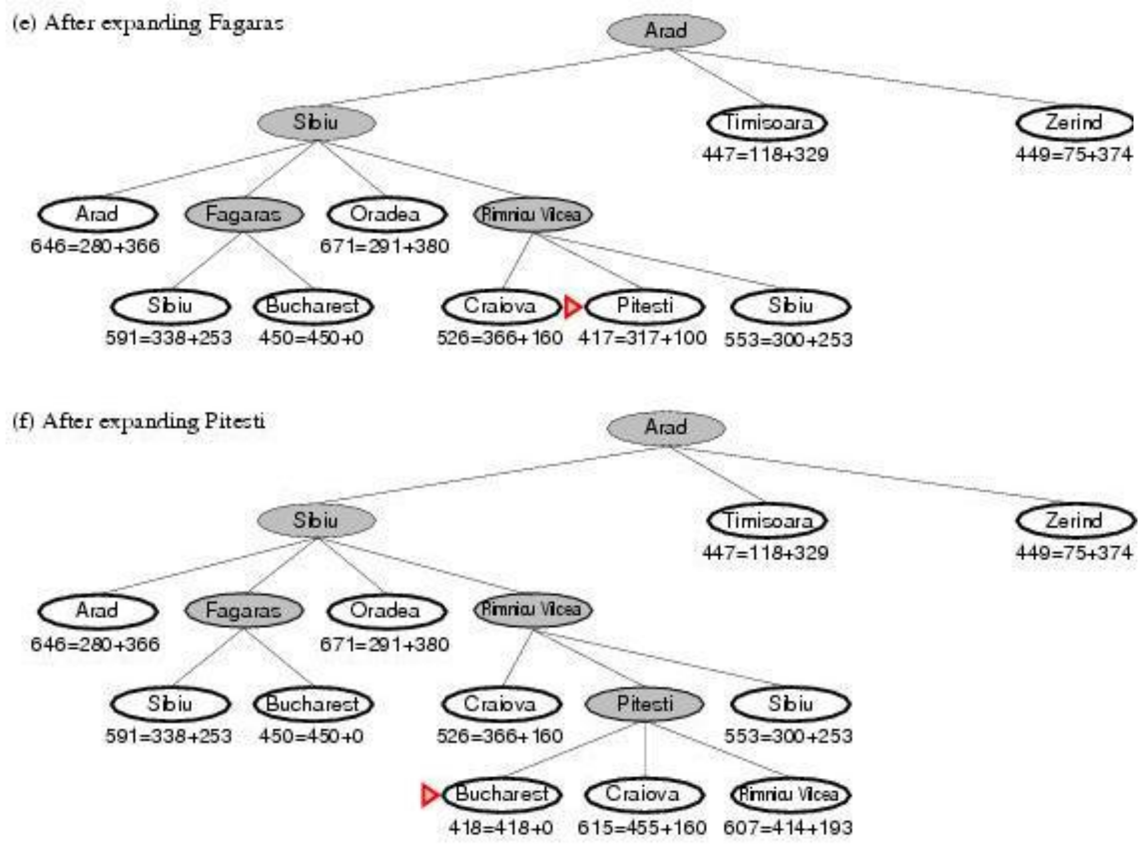


Figure 2.3 Stages in A* Search

Drawbacks

i) **A*search** keeps all generated nodes in memory (as do all GRAPH-SEARCH algorithms)

ii) A* usually runs out of space long before it runs out of time.

iii) A* is not practical for many large-scale problems and recently developed algorithms have overcome the space problem without sacrificing optimality or completeness, at a small cost in execution time.

Memory bounded heuristic search:

The simplest way to reduce memory requirements for A* is to adapt the idea of iterative deepening to the heuristic search context, resulting in the iterative-deepening A* (lDA*) algorithm.

The main difference between IDA* and standard iterative deepening is that the cutoff used is the *f-cost (g + h)* rather than the depth; at each iteration, the cutoff value is the smallest *f-cost* of any node that exceeded the cutoff on the previous iteration. IDA* is practical for many problems with unit step costs and avoids the substantial overhead associated with keeping a sorted queue of nodes.

2 memory bounded algorithm:

1) RBFS (recursive best-first search).

2) MA* (Memory-bounded A*) and SMA*(simplified memory MA*)

RBFS:

- It attempts to mimic the operation of BFS.
- Suffers from using too little memory.
- Even if more memory were available , RBFS has no way to make use of it.
- 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.

```
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 fcost 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 shows how RBFS reaches Bucharest.

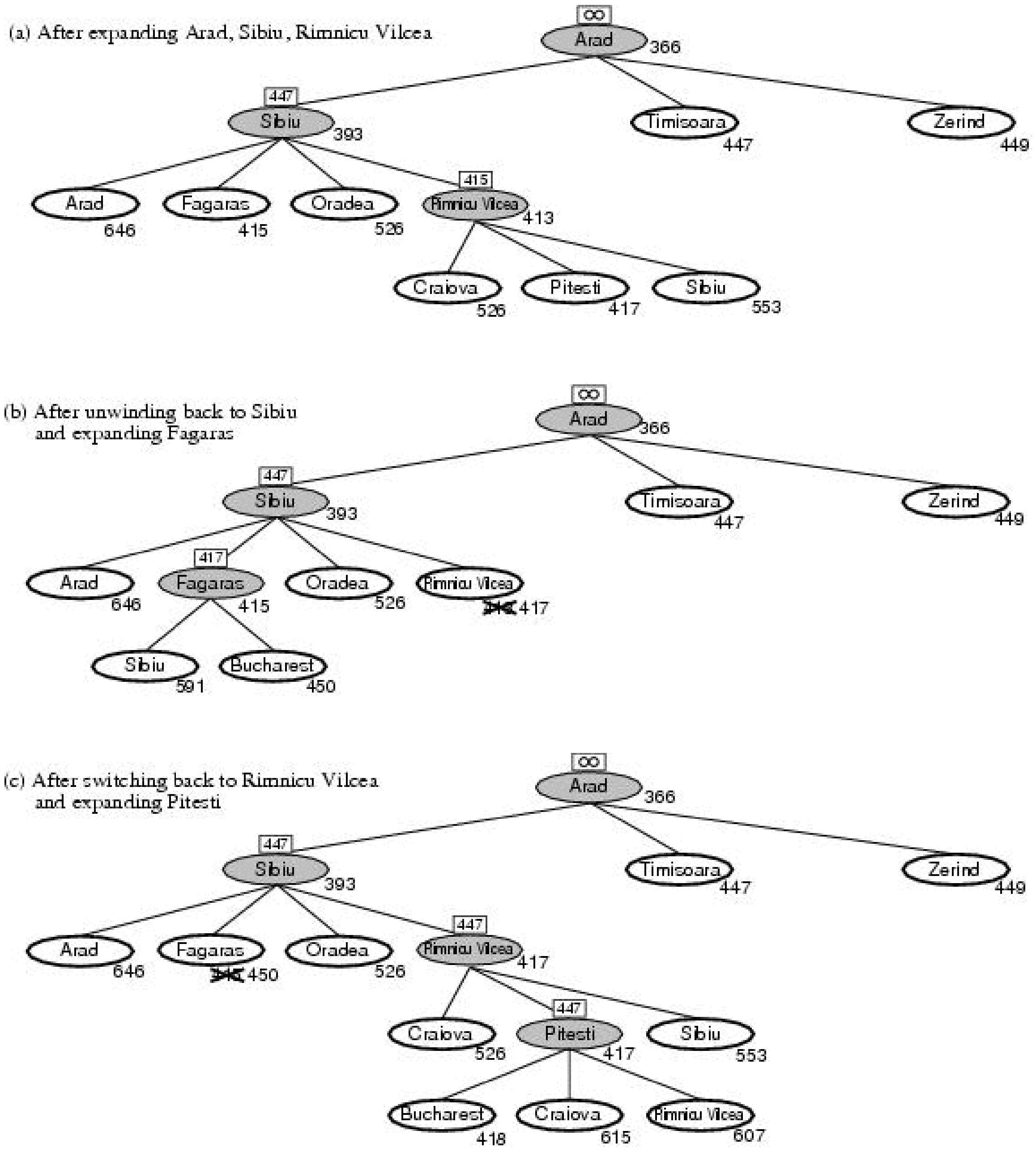


Figure 2.4 Stages in an RBFS search for the shortest route to Bucharest.

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* and RBFS suffer from using ***too little*** memory. Between iterations, IDA* retains only a single number: the current *I-cost* limit. REFS retains more information in memory, but it uses only *O(bd)* memory: even if more memory were available, RBFS has no way to make use of it. A Search technique which uses all available memory, two algorithms that do this

(1) MA* (memory-bounded A*) and

(2) SMA* (simplified MA*)

SMA*:

- Proceeds life A*,expands best leaf until memory is full.
- Cannot add new node without dropping an old one. (always drops worst one)
- Expands the best leaf and deletes the worst leaf.
- If all have same f-value-selects same node for expansion and deletion.
- SMA* is complete if any reachable solution.
- SMA* is optimal if any optimal solution is reachable; otherwise it returns the best reachable solution.SMA* might well be the best general-purpose algorithm for finding optimal solutions, particularly when the state space is a graph, step costs are not uniform, and node generation is expensive compared to the additional overhead of maintaining the open and closed lists.

Heuristic Function

A Heuristic technique helps in solving problems, even though there is no guarantee that it will never lead in the wrong direction. There are heuristics of every general applicability as well as domain specific. The strategies are general purpose heuristics. In order to use them in a specific domain they are coupler with some domain specific heuristics. There are two major ways in which domain - specific, heuristic information can be incorporated into rule-based search procedure.

Some different heuristics for the 8-puzzle problem shown in figure 2.5. A typical solution to the puzzle has around 20 steps. The branching factor is about 3. Hence, an exhaustive search to depth 20 will look at about $3^{20} = 3.5 \times 10^9$ states.

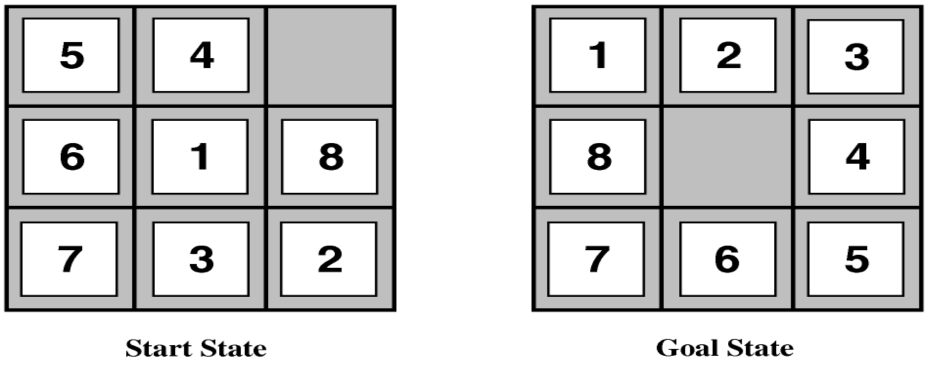


Figure 2.5 8 Puzzle Problem

By keeping track of repeated states, this number can be cut down to 9!=362,880 different arrangements of 9 squares. We need a heuristic to further reduce this number.If we want to find shortest solutions, we need a function that never overestimates the number of steps to the goal. Here are two possibilities:

- h1= the number of tiles that are in the wrong position. This is admissible because any tile that is out of place must be moved at least once.
- h2= the sum of the distances of the tiles from their goal positions. Since tiles cannot be moved diagonally, we use **city block distance**. h2 is also admissible.

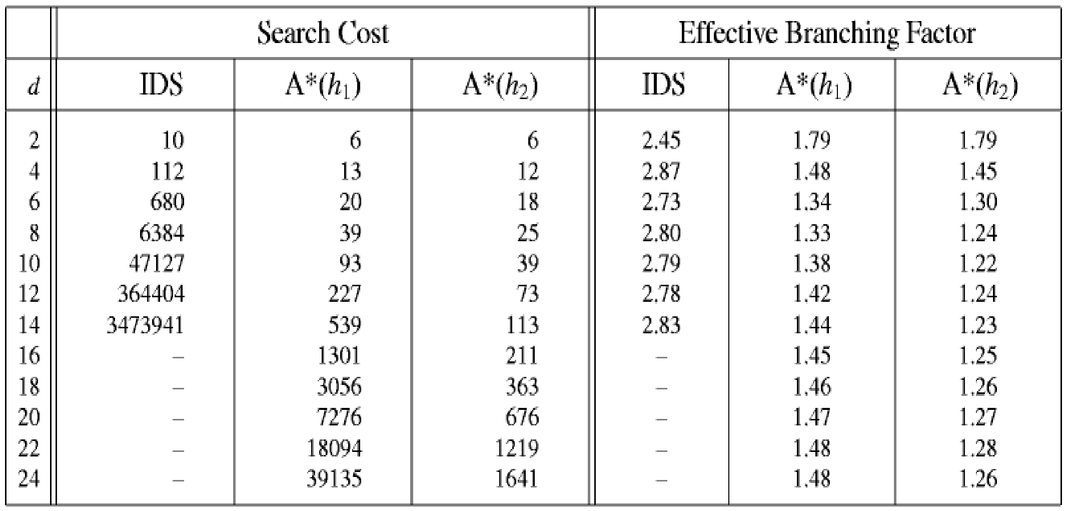
The effect of heuristic accuracy on performance

One way to characterize the quality of an heuristic is the **effective branching factor** b^*. If the total number of nodes expanded 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 nodes.

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

Example if A^* finds a solution at depth 5 using 52 nodes, the effective branching factor is 1.91. The effective branching factor of an heuristic is fairly constant over problem instances and, hence, experimental measurements of b^* on a small set of problems provides a good approximation of the heuristic's usefulness.

A well designed heuristic should have a value of b^* that is close to 1. The following is a table of the performance of A^* with h1 and h2 above on 100 randomly generated problems. It shows that h2 is better than h1 and that both are much better than 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	364404	227	73	2.78	1.42	1.24
14	3473941	539	113	2.83	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

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.

State Space Landscape

To understand local search, it is better explained using **state space landscape** as shown in figure 2.6.

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**.

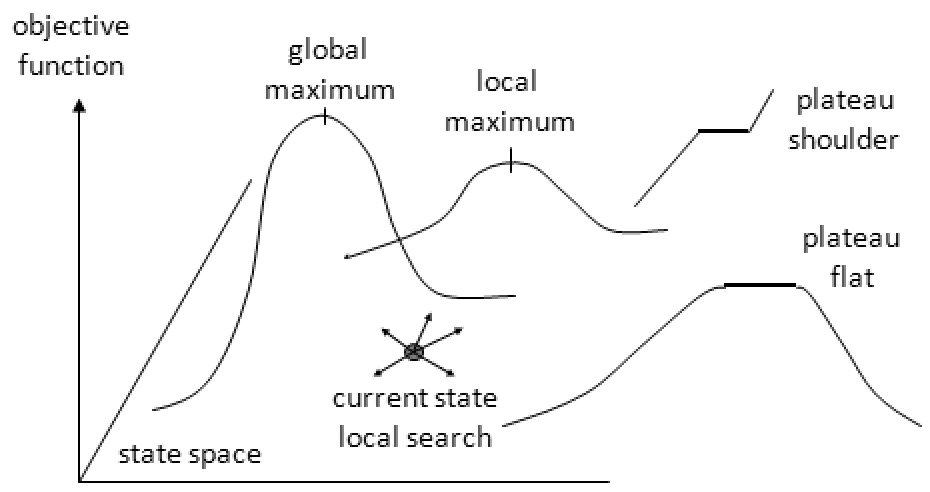
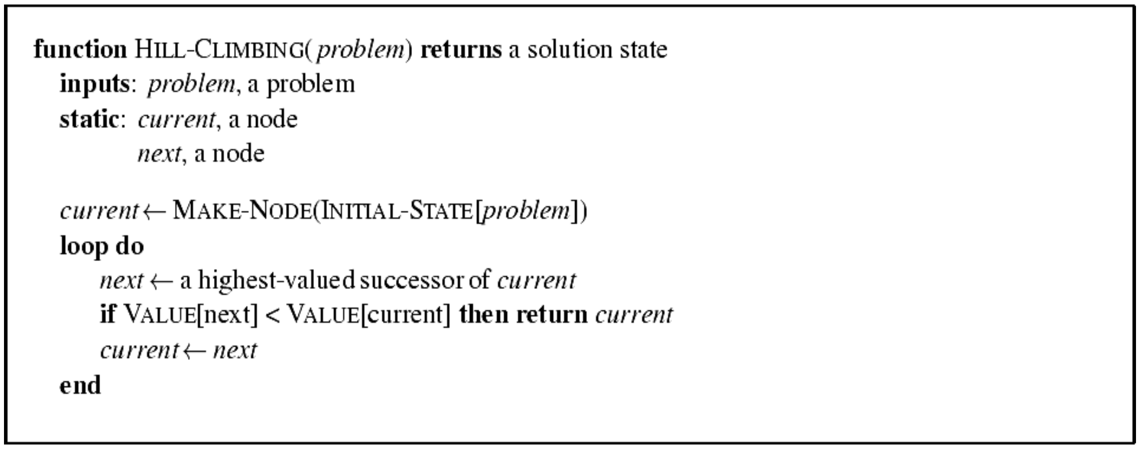


Figure 2.6 State Space Landscape

Hill Climbing

- The Hill-climbing search algorithm is a loop that continually moves in the direction of increasing value.
- The algorithm only records the state and its evaluation instead of maintaining a search tree. It takes a problem as an input, and it keeps comparing the values of the current and the next nodes.
- The next node is the highest-valued successor of the current node.
- If the value of the current node is greater than the next node, then the current node will be returned. Otherwise, it will go deeper to look at the next node of the next node.
- It 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.Hill climbing does not maintain a search tree, so the current node data structure need only record the state and its objective function value.
- Hill-climbing does not look ahead beyond the immediate neighbors of the current state.

The peaks are found on a surface of states where height is defined by Hill-climbing function.



```
function HILL-CLIMBING(problem) returns a solution state
  inputs: problem, a problem
  static: current, a node
          next, a node

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

Problems with hill-climbing

Hill-climbing often gets stuck for the following reasons.

i) Local maxima
ii) plateau

iii)Ridges

i) Local Maxima

Local maxima is a peak that is lower than the highest peak in the state space. When a local maxima is reached, the algorithm will halt even a solution has not been reached yet.

ii). Plateaux

A plateaux is an area of the state space where the neighbors are about the same height. In such a situation, a random walk will be generated.

iii). Ridges

A ridge may have steeply sloping sides towards the top, but the top only slopes gently towards a peak. In this case, the search makes little progress unless the top is directly reached, because it has to go back and forth from side to side.

It must be possible to encounter a situation that no further progress can be made from one certain starting point. If this happens, the Random restart hill-climbing is the obvious thing to do. As the name says, it randomly generates different starting points over again until it halts. It saves the best result found so far. And it can eventually find out the optimal solution if enough iteration are allowed.

As a matter of fact, and obviously, the fewer local maxima, the quicker it finds a good solution. But usually, a reasonably good solution can be found after a small number of iterations.

Variations of Hill Climbing

- Stochastic HC: chose randomly among the neighbors going uphill.
- First-choice HC: generate random successors until one is better. Good for states with high numbers of neighbors.
- Random restart: the sideway moves restart from a random state.
- Evolutionary hill-climbing: represents potential solutions as strings and performs random mutations Keeps the mutations that are better states. It's a particular case of first-choice and the ancestor of the genetic algorithms.

Simulated Annealing

- The simulated annealing takes some downhill steps to escape the local maxima, and it picks random moves instead of picking the best move.

- If the move actually improves the situation, it will keep executing the move. Otherwise, it will make the moves of a probability less than one.

- When the end of the searching is close, it starts behaving like hill-climbing.

- The word "annealing" is originally the process of cooling a liquid until it freezes. The Simulated-annealing function takes a problem and a schedule as inputs Here, schedule is a mapping determining how fast the temperature should be lowered.

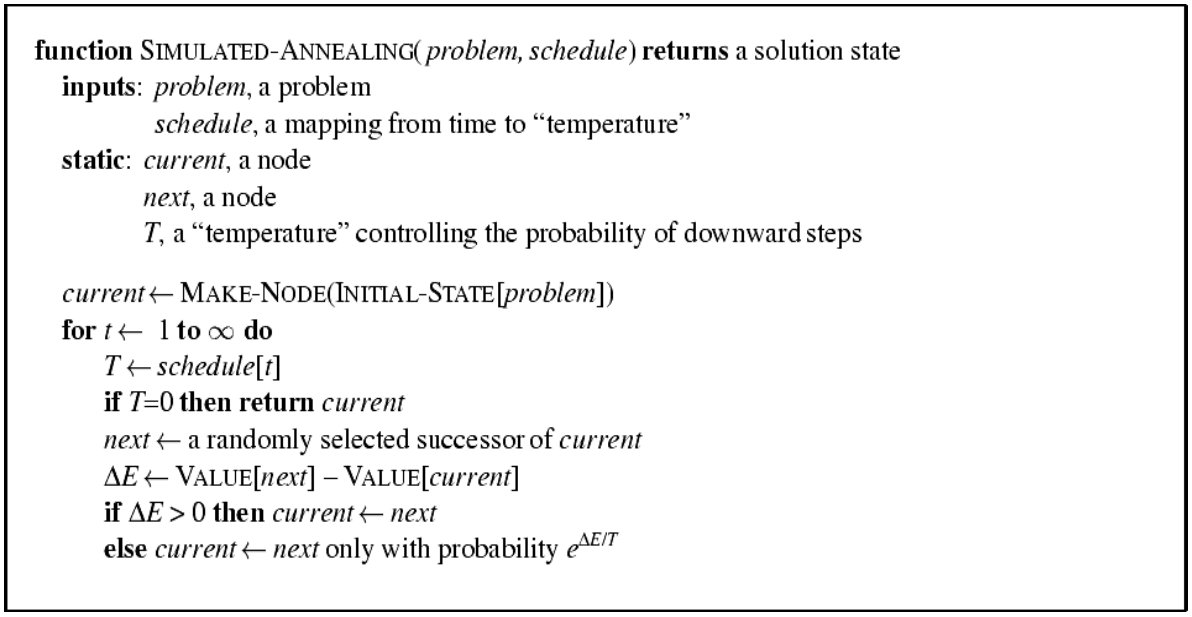
- Again, the algorithm keeps comparing the values of the current and the next nodes, but here, the next node is a randomly selected successor of the current node.

- It also maintains a local variable T which is the temperature controlling the probability of downward steps.

- By subtracting the values of the current node from the next node to obtained the difference Delta-E, the algorithm can determine the probability of the next move. If Delta-E is greater than zero, then the next node will be looked at.

- Otherwise, the probability for the next node to be looked at is e to the power Delta-E over T.

- In other word, the variable Delta-E is actually the amount by which the evaluation is worsened



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"
  static: current, a node
          next, a node
          T, a "temperature" controlling the probability 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
      $\Delta E$ ← VALUE[next] – VALUE[current]
      if $\Delta E > 0$ then current ← next
      else current ← next only with probability $e^{\Delta E/T}$
```

Applications of Simulated Annealing:

- Traveling sales man problem.
- VLSI Design.
- Production Scheduling.
- Timetable problem.
- Image Processing.

Local beam search

Keeping just one node in memory might seem to be an extreme reaction to the problem of memory limitations. The local beam search algorithm 10 keeps track of *k* states rather than just one. It begins with *k* randomly generated states. At each step, all the successors of all *k* states are generated. If anyone is a goal, the algorithm halts. Otherwise, it selects the *k* best successors from the complete list and repeats.

A local beam search with *k* states runs *k* random restarts in parallel instead of in sequence. In a random-restart search, each search process runs independently of the others. In a local beam search, useful information is passed among the k parallel search threads.

Stochastic Beam Search

This search helps to avoid the above problem. Instead of choosing the best k from the pool of successor stochastic beam search chooses k successors at random, with the probability that choosing a successor is an increasing function of its value**.**

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, GA 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-queens state must specify the positions of 8 queens, each in a column of 8 squares, and so requires 8 x $\log_2 8$ = 24 bits.

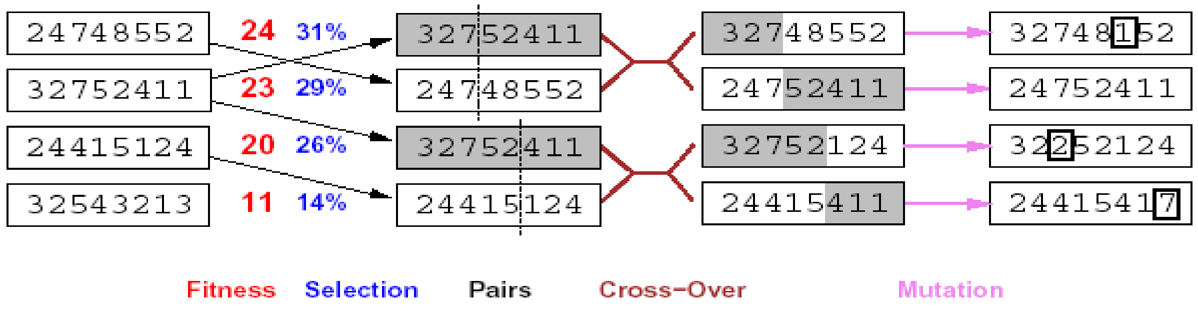


Figure 2.7 Genetic Algorithm

Fitness function : In figure 2.7 each state is rated by the evaluation function or the **fitness function**. This function should return higher values for better states so, for 8 queens pbm, we use number of non attacking pairs of queens, which has a value of 28 for a solution. The value of the four states are 24,23,20,22.

Selection : In Figure 2.7 ,a random choice of two pairs is selected for reproduction, in accordance with the probabilities in figure 2.7. One individual is selected twice and not at all once.

Cross over : For each pair to be mated, a crossover point is randomly chosen from the positions in the string. In Figure 2.7 the crossover points are after the third digit in the first pair and after the fifth digit in the second pair. The first child of the first pair gets the first three digits from the first parent and the remaining digits from the second parent.

Mutation : Each location is subject to random **mutation** with a small independent probability. One digit was mutated in the first, third, and fourth offspring. In the 8-queens problem, this corresponds to choosing a queen at random and moving it to a random square in its column.

Advantages of GA

Advantage comes from the ability of crossover to combine large blocks of letters that have evolved independently to perform useful functions, thus raising the level of granularity at which the search operates.

Schema

Substring in which some of the positions can be left unspecified is called Schema.

Strings that match the schema (such as 24613578) are called instances of the schema.

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
inputs: population, a set of individuals
FITNESS-FN, a function that measures the fitness of an individual
repeat
new.populatioti +- 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 .populaiion
until some individual is fit enough, or enough time has elapsed
return the best individual in population, according to FITNESS-FN
```

```
function REPRODUCE(X, y) returns an individual
inputs: x, y, parent individuals
n +- LENGTH(X)
c +- random number from 1 to n
return ApPEND(SUBSTRING(X,1, c), SUBSTRING(y, c + 1, n))
```

Constraint Specification Problem

- 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_i of possible values. Each constraint C_i involves some subset of the 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 so- lution to a CSP is a complete assignment that satisfies all the constraints.
- Some CSPs also require a solution that maximizes an objective function.

Suppose that, having tired of Romania, we are looking at a map of Australia showing each of its states and territories, and that we are given the task of coloring each region either red, green, or blue in such a way that no neighboring regions have the same color. To formulate this as a CSP, we define the variables to be the regions shown in figure 2.8: 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)}. 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.

The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem. The goal is to assign colors to each region so that no neighboring regions have the same color. The map coloring problem represented as a constraint graph.

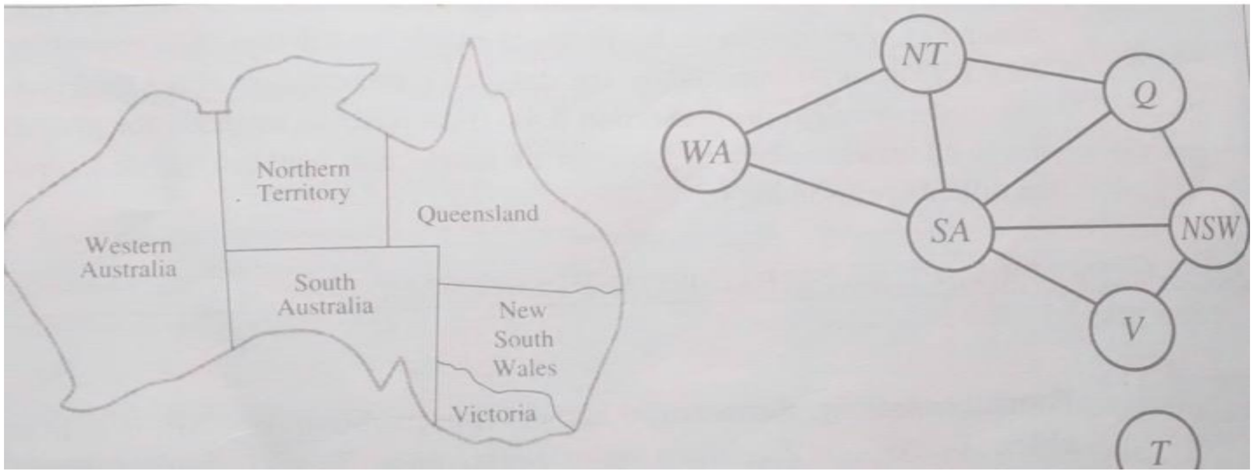


Figure 2.8 Road Map

CSP can be given an incremental formulation 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 for every step.

Examples of constraint satisfaction problems:

Example 1: The n-Queen problem: The local condition is that no two queens attack each other, i.e. are on the same row, or column, or diagonal.

Example 2: A crossword puzzle: We are to complete the puzzle

```
     1   2   3   4   5
   +---+---+---+---+---+        Given the list of words:
 1 | 1 |   | 2 |   | 3 |        AFT       LASER
   +---+---+---+---+---+                  ALE       LEE
 2 | # | # |   | # |   |        EEL       LINE
```

```
  +---+---+---+---+---+               HEEL    SAILS
3 | # | 4 |   | 5 |   |         HIKE  SHEET
  +---+---+---+---+---+               HOSES   STEER
4 | 6 | # | 7 |   |   |         KEEL  TIE
  +---+---+---+---+---+               KNOT
5 | 8 |   |   |   |   |
  +---+---+---+---+---+
6 |   | # | # |   | # |     The numbers 1,2,3,4,5,6,7,8 in the crossword
  +---+---+---+---+---+         puzzle correspond to the words
                                      that will start at those locations.
```

Example 3: A cryptography problem: In the following pattern

```
      S E N D
      M O R E
  =========
  M O N E Y
```

We have to replace each letter by a distinct digit so that the resulting sum is correct.

Example 4: A map coloring problem: We are given a map, i.e. a planar graph, and we are told to color it using three colors, green, red, and blue, so that no two neighboring countries have the same color.

All these examples are instances of the same pattern, captured by the following definition:

A Constraint Satisfaction Problem is characterized by:

- a set of variables {x1, x2, .., xn},
- for each variable xi a domain Di with the possible values for that variable, and
- a set of constraints, i.e. relations, that are assumed to hold between the values of the variables. [These relations can be given intentionally, i.e. as a formula, or extensionally, i.e. as a set, or procedurally, i.e. with an appropriate generating or recognising function.] We will only consider constraints involving one or two variables.

The constraint satisfaction problem is to find, for each i from 1 to n, a value in Di for xi so that all constraints are satisfied.

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.

ALGORITHM

```
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
```

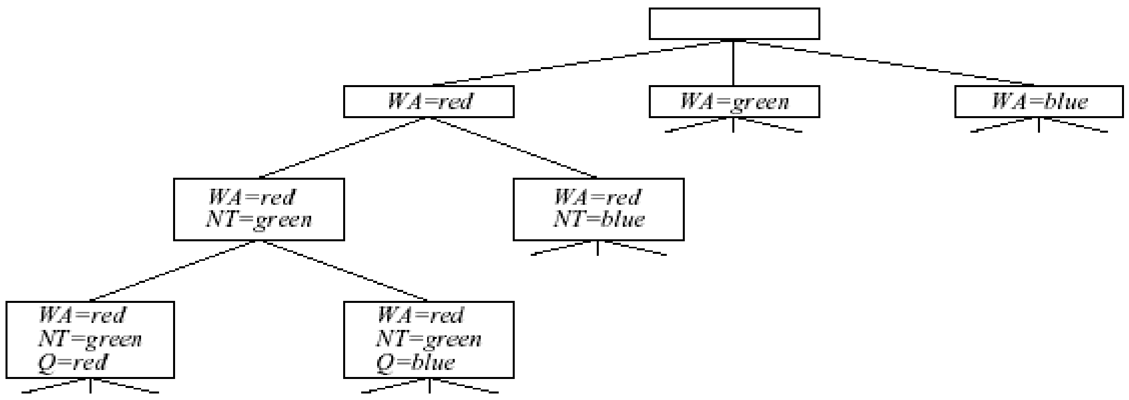


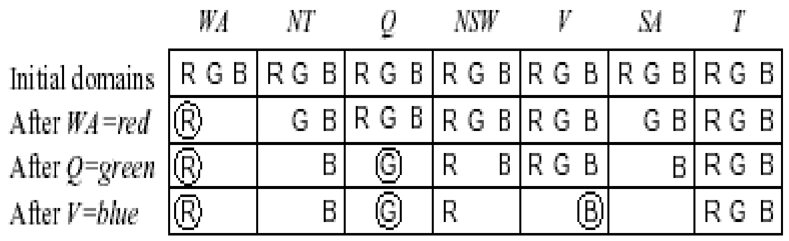
Figure 2.9 Simple backtracking for map colouring

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.



	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*	Ⓡ	G B	R G B	R G B	R G B	G B	R G B
After *Q=green*	Ⓡ	B	Ⓖ	R B	R G B	B	R G B
After *V=blue*	Ⓡ	B	Ⓖ	R	Ⓑ		R G B

Figure 2.10 Map coloring with forward checking

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

It provides a fast method of constraint propagation that is substantially stronger than forward checking. Here arc refers to a directed arc in the constraint graph. Such as the arc from SA to NSW. Given the current domains of SA and NSW,the arc is consistent if, for *every* value x of SA, there is *some* value y of NSW that is consistent with x. In the third row the current domains of SA and NSW are fblueg and fred; blueg respectively. For SA=blue, there is a consistent assignment for NSW, namely, NSW =red; therefore, the arc from SA to NSW is consistent. On the other hand, the reverse arc from NSW to SA is not consistent: for the assignment NSW =blue, there is no consistent assignment for SA. The arc can be made consistent by deleting the value blue from the domain of NSW.

We can also apply arc consistency to the arc from SA to NT at the same stage in the search process. The third row of the table shows that both variables have the domain fblueg. The result is that blue must be deleted from the domain of SA, leaving the domain empty. Thus, applying arc consistency has resulted in early detection of an inconsistency that is not detected by pure forward checking.

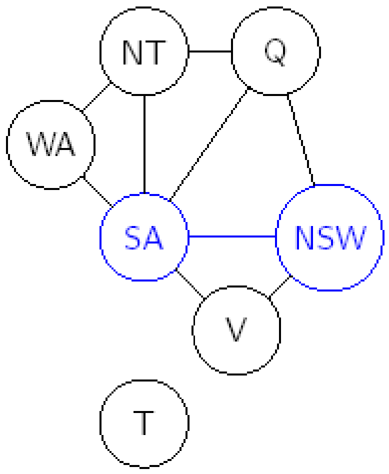


Figure 2.11 Arc consistency

```
function AC-3(csp) returns the CSP, possibly with reduced domains
inputs: csp, a binary CSP with variables fX1; X2; : : : ; Xng
local variables: queue, a queue of arcs, initially all the arcs in csp
while queue is not empty do
(Xi; Xj) REMOVE-FIRST(queue)
if REMOVE-INCONSISTENT-VALUES(Xi; Xj ) then
for each Xk in NEIGHBORS[Xi] do
add (Xk; Xi) to queue
function REMOVE-INCONSISTENT-VALUES(Xi; Xj ) returns true iff we remove a value
removed  false
```

```
for each x in DOMAIN[Xi] do
if no value y in DOMAIN[Xj ] allows (x ,y) to satisfy the constraint between Xi and Xj
then delete x from Domain[Xi]
return removed
```

Intelligent backtracking: looking backward

The BACKTRACKING-SEARCH algorithm has a very simple policy for what to do when a branch of the search fails: back up to the preceding variable and try a different value for it. This is called **chronological backtracking**, because the *most recent* decision point is revisited. In this subsection, we will see that there are much better ways.

Consider what happens when we apply simple backtracking with a fixed variable ordering Q, NSW, V , T, SA, WA, NT. Suppose we have generated the partial assignment fQ=red;NSW =green; V =blue; T =redg. When we try the next variable, SA, we see that every value violates a constraint. We back up to T and try a new color for Tasmania! Obviously this is silly—recoloring Tasmania cannot resolve the problem with South Australia.

A more intelligent approach to backtracking is to go all the way back to one of the CONFLICT SET set of variables that *caused the failure*. This set is called the **conflict set**; here, the conflict set for SA is fQ;NSW; V g. In general, the conflict set for variable X is the set of previously assigned variables that are connected to X by constraints. The **backjumping** method backtracks to the *most recent* variable in the conflict set

LOCAL SEARCH FOR CSPS

- Local search using the min-conflicts heuristic has been applied to constraint satisfaction problems with great success. They use a complete-state formulation: the initial state assigns a value to every variable, and the successor function usually works by changing the value of one variable at a time.
- In choosing a new value for a variable, the most obvious heuristic is to select the value that results in the minimum number of conflicts with other variables—the min-conflicts heuristic.

- Min-conflicts is surprisingly effective for many CSPs, particularly when given a reasonable initial state. Amazingly, on then-queens problem, if you don‘t count the initial placement of queens, the runtime of minconflicts is roughly independent of problem size

```
function MIN-CONFLICTS(csp, max steps) returns a solution or failure
  inputs: csp, a constraint satisfaction problem
    max steps, the number of steps allowed before giving up
current an initial complete assignment for csp

for i = 1 to max steps do

        if current is a solution for csp then return current
        var a randomly chosen, conflicted variable from VARIABLES[csp]
        value the value v for var that minimizes CONFLICTS(var, v, current, csp)
        set var =value in current

return failure
```

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 decomposotion into subproblems.

The algorithm has the following steps:

1. Choose any variable as the root of the tree, and order the variables from the root to the leaves in such a way that every node’s parent in the tree precedes it in the ordering. (See
Figure 2.12(b).) Label the variables X1; : : : ;Xn in order. Now, every variable except the root has exactly one parent variable.

2. For j from n down to 2, apply arc consistency to the arc (Xi;Xj), where Xi is the parent of Xj , removing values from DOMAIN[Xi] as necessary.

3. For j from 1 to n, assign any value for Xj consistent with the value assigned for Xi,

- where Xi is the parent of Xj .

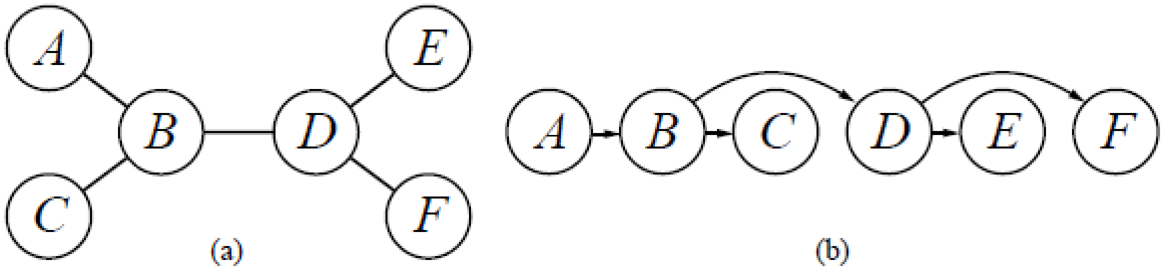


Figure 2.12 Constraint graph of a tree structured CSP

There are two key points to note. First, after step 2 the CSP is directionally arc-consistent,so the assignment of values in step 3 requires no backtracking. Second, by applying the arc-consistency checks in reverse order in step 2, the algorithm ensures that any deleted values cannot endanger the consistency of arcs that have been processed already. The complete algorithm runs in time $O(nd^2)$.

Tree decomposition

Each subproblem is solved independently, and the resulting solutions are then combined. Like most divide-and-conquer algorithms, this works well if no subproblem is too large. Figure 2.13 shows a tree decomposition of the mapcoloring problem into five subproblems. A tree decomposition must satisfy the following three requirements:

_ Every variable in the original problem appears in at least one of the subproblems.

_ If two variables are connected by a constraint in the original problem, they must appear together (along with the constraint) in at least one of the subproblems.

_ If a variable appears in two subproblems in the tree, it must appear in every subproblem along the path connecting those subproblems.

The first two conditions ensure that all the variables and constraints are represented in the decomposition. The third condition seems rather technical, but simply reflects the constraint that any given variable must have the same value in every subproblem in which it appears; the links joining subproblems in the tree enforce this constraint. For example, SA appears in all four of the connected subproblems in Figure 2.13

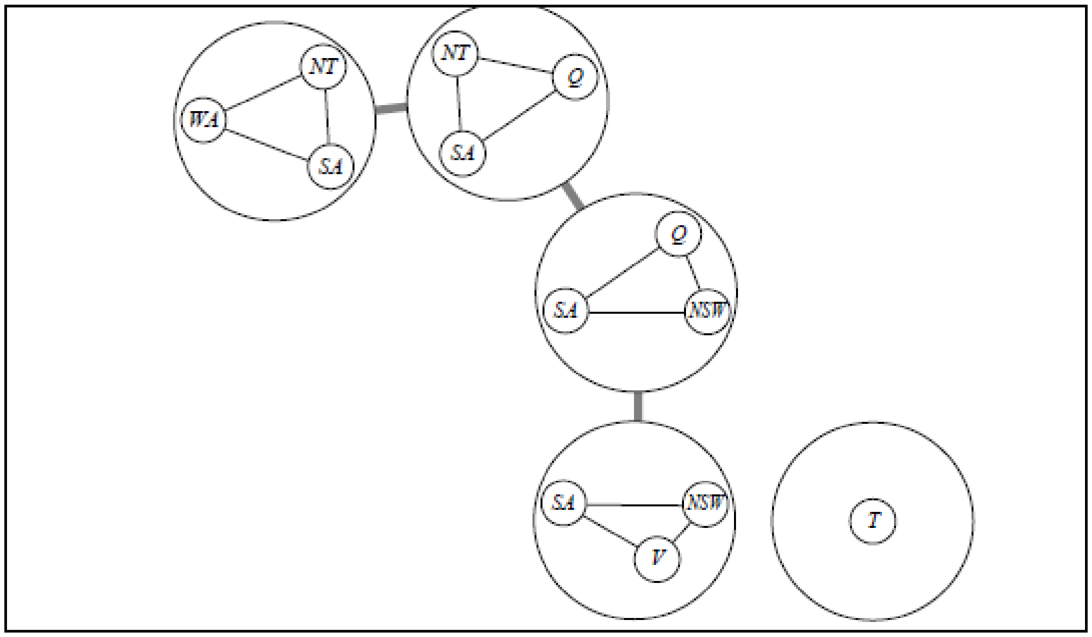


Figure 2.13 Tree Decomposition

ADVERSARIAL SEARCH

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

GAMES

- Mathematical Game Theory, a branch of economics, views any multi agent 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.

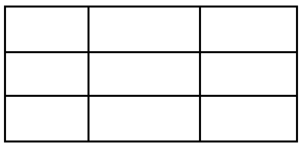
Optimal Decisions in Game

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. His 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 shows the part of the game tree for tic-tac-toe (nougats 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.

x	0	x
	0	x
	0	

Won by MIN

x	0	x
0	0	x
x	x	0

Draw between MIN and MAX

x	0	x
	x	x
x	0	0

Won by MAX

- The 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. The below figure 2.14 show about the search tree for the game of tic-tac-toe.

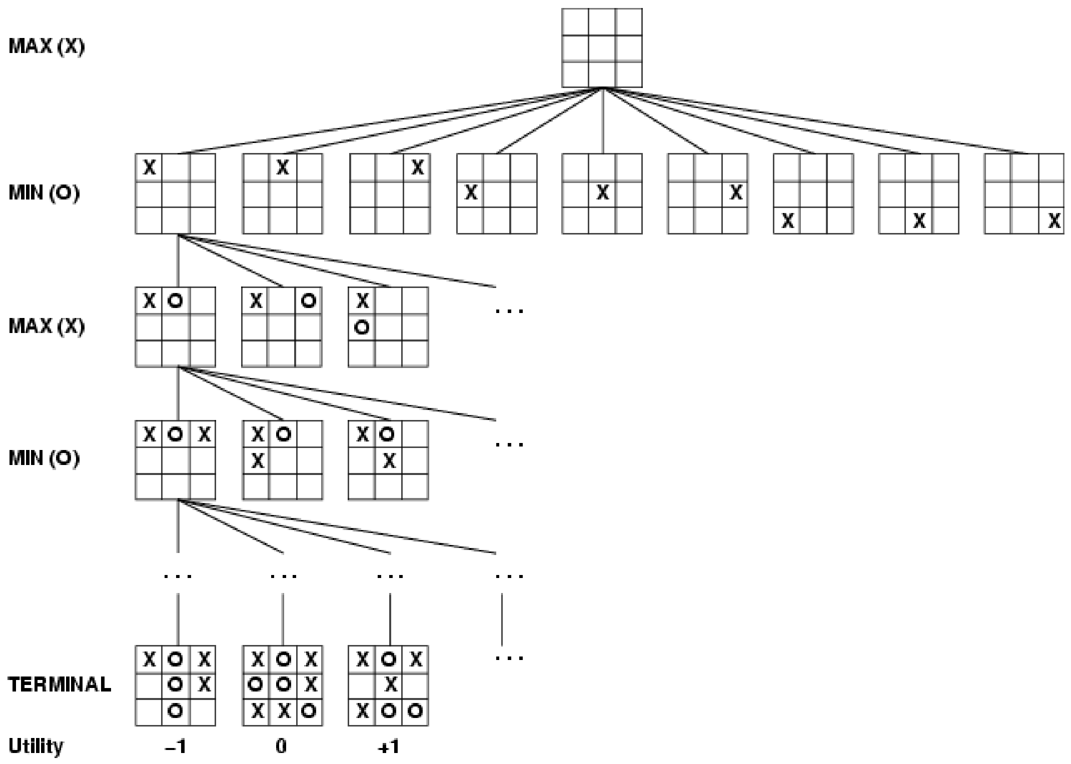


Figure 2.14 search tree for the game of tic-tac-toe.

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.

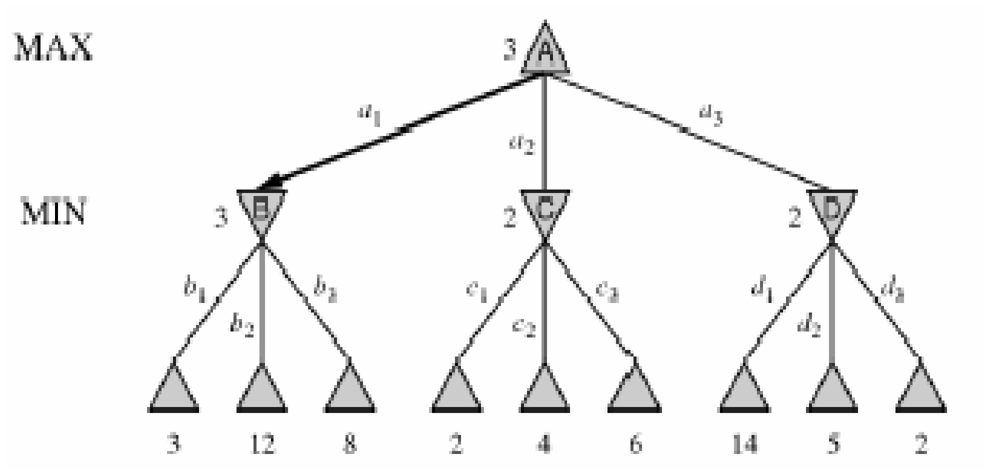


Figure 2.15 Two ply game tree

Figure 2.15 shows A Two ply game tree. The nodes are “MAX nodes”, in which it is MAX’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.

```
function MINIMAX-DECISION(state) returns an action
    return arg max$_{a \in \text{ACTIONS}(s)}$ MIN-VALUE(RESULT(state, a))
```

```
function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    $v \leftarrow -\infty$
    for each a in ACTIONS(state) do
        $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s, a)))
    return v
```

```
function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    $v \leftarrow \infty$
    for each a in ACTIONS(state) do
        $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s, a)))
    return v
```

The minimax algorithm 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 the previous fig ,the algorithm first recourses down to the three bottom left nodes, and uses the utility 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.

ALPHA-BETA PRUNING

Pruning: The process of eliminating a branch of the search tree from consideration without examining is called pruning. The two parameters of pruning technique are:

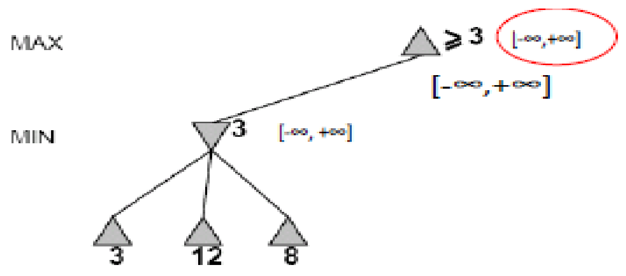
1.**Alpha (α)**: Best choice for the value of MAX along the path or lower bound on the value that on maximizing node may be ultimately assigned.

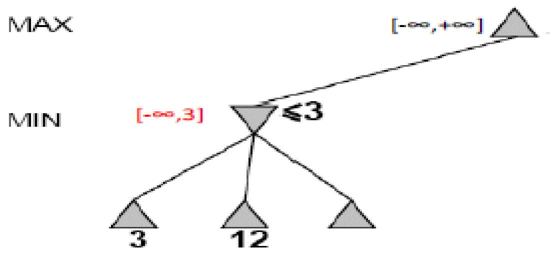
2.**Beta (β):** Best choice for the value of MIN along the path or upper bound on the value that a minimizing node may be ultimately assigned.

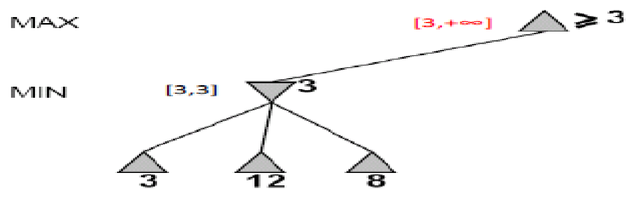
Alpha-Beta Pruning: The alpha and beta values are applied to a minimax tree, it returns the same move as minimax, but prunes away branches that cannot possibly influence the final decision is called Alpha-Beta pruning or Cutoff.

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.

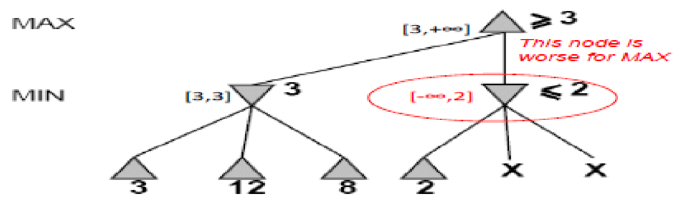
Consider the two ply game tree from figure 2.16 . The different Stage of the calculation for optimal decision for the game tree.

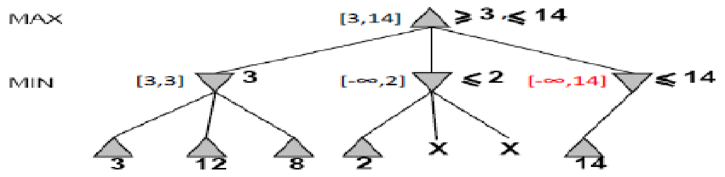


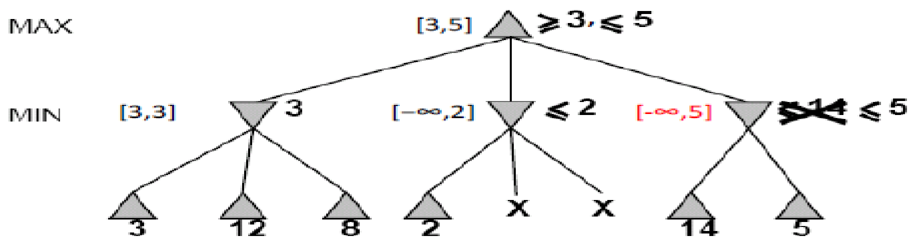


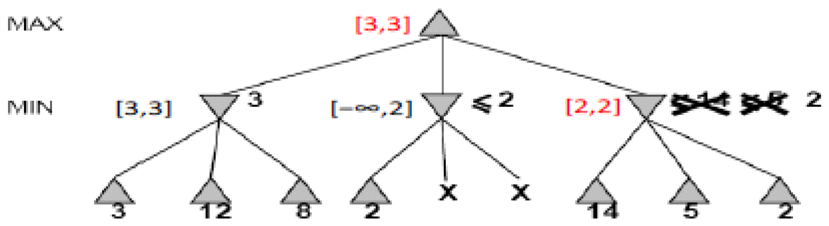


Alpha-Beta Example (continued)









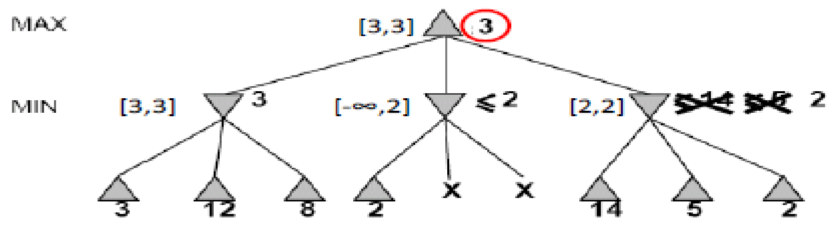


Figure 2.16 Alpha Beta Pruning

Stages in the calculation of the optimal decision for the game tree in Figure 2.16. At each point, we show the range of possible values for each node.

(a) The first leaf below *B* has the value 3. Hence, *B,* which is a MIN node, has a value of *at most* 3.

(b) The second leaf below *B* has a value of 12; MIN would avoid this move, so the value of *B* is still at most 3.

(c) The third leaf below *B* has a value of 8; we have seen all *B's* successors, so the value of *B* is exactly 3. Now, we can infer that the value of the root is *at least* 3, because MAX has a choice worth 3 at the root.

(d) The first leaf below C has the value 2. Hence, C, which is a MIN node, has a value of *at most* 2. But we know that *B* is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other successors of C. This is an example of alpha-beta pruning.

(e) The first leaf below *D* has the value 14, so *D* is worth *at most* 14. This is still higher than MAX'S best alternative (i.e., 3), so we need to keep exploring *D's* successors. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14.

(f) The second successor of *D* is worth 5, so again we need to keep exploring. The third successor is worth 2, so now *D* is worth exactly 2. MAX'S decision at the root is to move to *B*, giving a value of 3

Consider a node *n* in the tree ---

• If player has a better choice at:

– Parent node of n

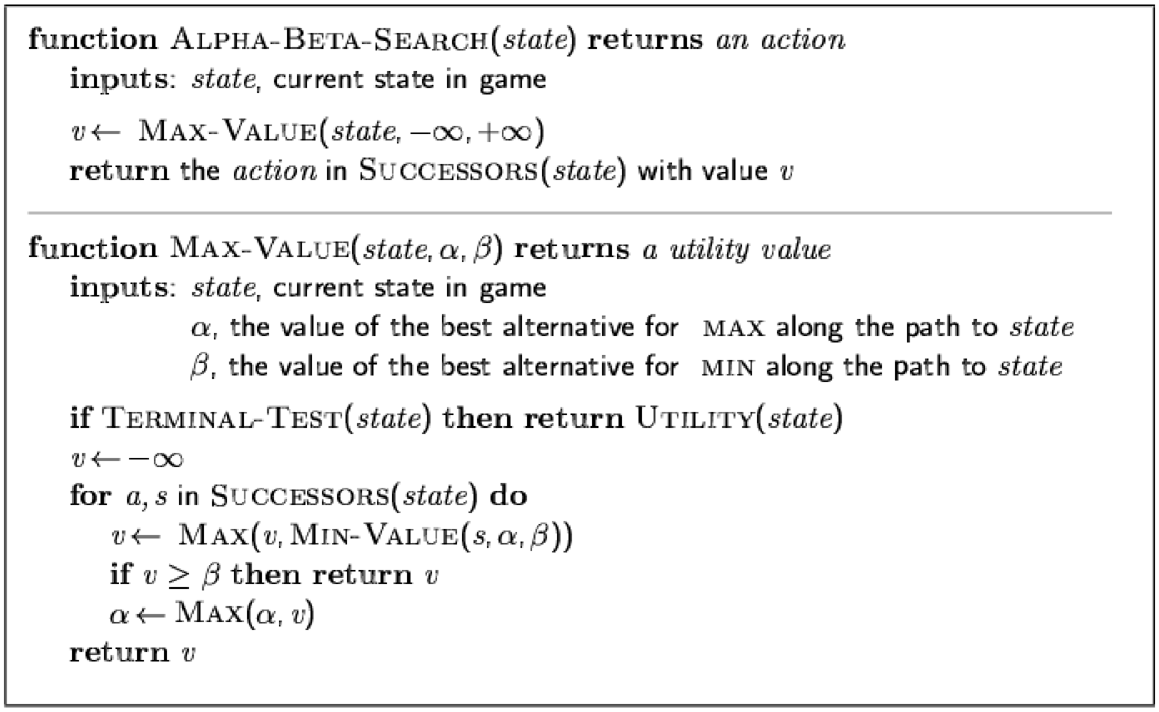
– Or any choice point further up

• Then *n* will never be reached in play.

• Hence, when that much is known about *n*, it can be pruned.

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.

Alpha –beta pruning can be applied to trees of any depth, and it is often possible to prune entire sub trees rather than just leaves.



```
function ALPHA-BETA-SEARCH(state) returns an action
    inputs: state, current state in game
    $v \leftarrow$ MAX-VALUE(state, $-\infty, +\infty$)
    return the action in SUCCESSORS(state) with value $v$

function MAX-VALUE(state, $\alpha, \beta$) returns a utility value
    inputs: state, current state in game
            $\alpha$, the value of the best alternative for MAX along the path to state
            $\beta$, the value of the best alternative for MIN along the path to state
    if TERMINAL-TEST(state) then return UTILITY(state)
    $v \leftarrow -\infty$
    for $a, s$ in SUCCESSORS(state) do
        $v \leftarrow$ MAX($v$, MIN-VALUE($s, \alpha, \beta$))
        if $v \geq \beta$ then return $v$
        $\alpha \leftarrow$ MAX($\alpha, v$)
    return $v$
```

```
function MIN-VALUE($state, \alpha, \beta$) returns a utility value
  inputs: state, current state in game
          $\alpha$, the value of the best alternative for MAX along the path to state
          $\beta$, the value of the best alternative for MIN along the path to state

  if TERMINAL-TEST(state) then return UTILITY(state)
  $v \leftarrow +\infty$
  for $a, s$ in SUCCESSORS(state) do
    $v \leftarrow$ MIN($v$, MAX-VALUE($s, \alpha, \beta$))
    if $v \leq \alpha$ then return $v$
    $\beta \leftarrow$ MIN($\beta, v$)
  return $v$
```

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 at least 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 non terminal 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.

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.
Evaluation function Design :-
(i) The evaluation function should order the *terminal* states in the same way as the true utility function;
(ii)The computation must not take too long.

(iii) For non terminal states, the evaluation function should be strongly correlated with the actual chances of winning.

Example:

In chess, each material has its own value, that is called Material Value (i.e. each pawn is worth 1, a knight or bishop is worth 3, a rook 5, and the queen 9). Other features such as "good pawn structure" and "king safety" might be worth half a pawn, say. These feature values are then simply added up to obtain the evaluation of the position. A secure advantage equivalent to a pawn gives a substantial likelihood of winning, and a secure advantage equivalent to three pawns should give almost certain victory.

Mathematically, this kind of evaluation function is called a **weighted linear function,** because it can be expressed as *n*

EVAL(s) = *wlh(s)* + *w2h(s)* + ... + *wnfn('.,)* $=\sum_{i=1}^{n} wifi(s)$,

where each *Wi* is a weight and each *fi* is a feature of the position. For chess, the *fi* could be the numbers of each kind of piece on the board, and the *Wi* could be the values of the pieces

Cutting off search

To perform a cut-off test, an evaluation function should be applied to positions that are quiescent – unlikely to exhibit wild swings in value in the search tree.

The next step is to modify ALPHA-BETA-SEARCH so that it will call the heuristic EVAL function when it is appropriate to cut off the search.

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

We also must arrange for some bookkeeping so that the current *depth* is incremented on each recursive call. The most straightforward approach to controlling the amount of search is to set a fixed depth limit, so that CUTOFF-*TEST(state, depth)* returns *true* for all *depth* greater than some fixed depth *d.* (It must also return *true* for all terminal states, just as TERMINAL-TEST did.) The depth *d* is chosen so that the amount of time used will not exceed what the rules of the game allow.

QUIESCENCE :

It is restricted to consider only certain types of moves, such as capture moves, that will quickly resolve the uncertainties in the position.

HORIZON EFFECT:

It arises when the program is facing a move by the opponent that causes serious damage and is ultimately unavoidable.

Over the Search Horizon :

Move to a place where it cannot be detected.

Singular Extension :

It will avoid the horizon effect without adding too much search cost. It is a move that is clear than all other moves in a given position. It‘s branching factor is 1.

Forward Pruning :

It means some moves at a given node are pruned immediately without further consideration.

Games that include an element of 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. For example, white has rolled a 6-5, and has four possible moves shown in figure 2.17 . White knows what his or her own legal moves are, White doesnot know what black is going to roll and thus does not know what balck legal moves will be.

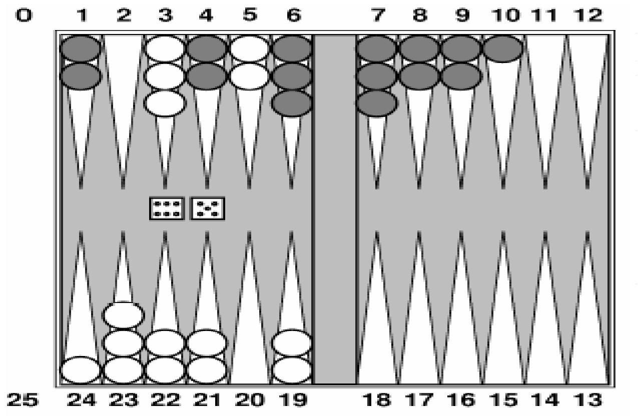


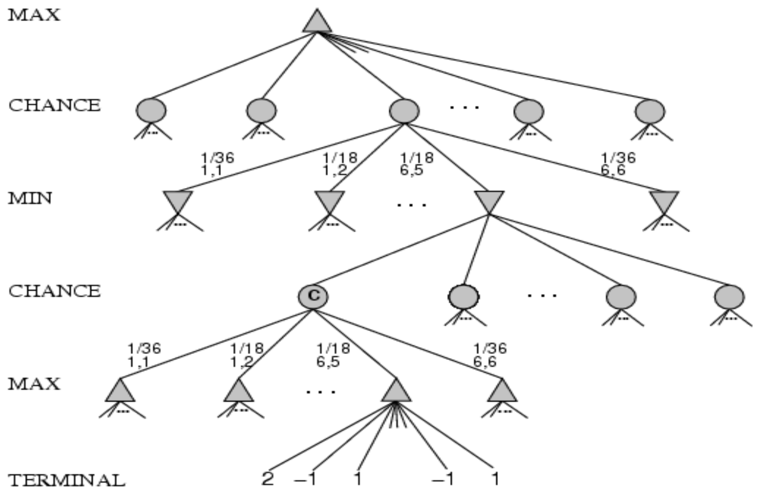
Figure 2.17 Backgammon Position

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)

A game tree in backgammon must include chance nodes in addition to MAX and MIN nodes. In the below figure chance nodes are shown as circles. The branches leading from each chance node denotes the possible dice rolls, and each is labeled with the roll and the chance node denote the possible dice rolls.



Expected minimax value

EXPECTED-MINIMAX-VALUE(n)=

UTILITY(n) If n is a terminal

$\max_s \in$ successors(n) MINIMAX-VALUE(s) If n is a max node

$\min_s \in$ successors(n) MINIMAX-VALUE(s) If n is a max node

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

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

Position evaluation in games with chance nodes

- Due to the presence of chance nodes means that one has to be more careful about what the evaluation values mean.

- The program behaves totally differently if there is a change in the scale of some evaluation values.

- The evaluation function must be a *positive linear* transformation of the probability of winning from a position

Complexity of Expectiminimax

If the program knew in advance all the dice rolls that would occur for the rest of the game, solving a game with dice would be just like solving a game without dice, which minimax does in *O(bm)* time. Because expectiminimax is also considering all the possible dice-roll sequences, it will take 0 *(bmnm),* where *ti* is the number of distinct rolls, d is the depth. Card game is also an example that takes an element of chance.