UNIT-II
SEARCHING TECHNIQUES

INFORMED SEARCH AND EXPLORATION :-

Informed(Heuristic) Search Strategies

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

Best-first search

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

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

Heuristic functions

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

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

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

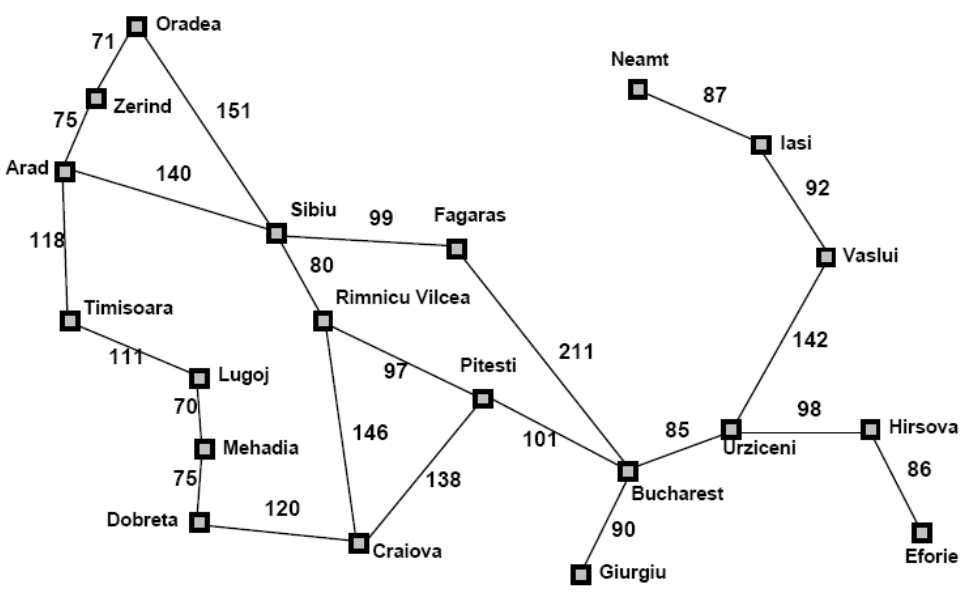
For example,in Romania,one might estimate the cost of the cheapest path from Arad to Bucharest via a **straight-line distance** from Arad to Bucharest(Figure 2.1).

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

Greedy Best-first search :-

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

- It evaluates the nodes by using the heuristic function f(n) = h(n).
- Taking the example of **Route-finding problems** in Romania , the goal is to reach Bucharest starting from the city Arad. We need to know the straight-line distances to Bucharest from various cities as shown in Figure 2.1. For example, the initial state is In(Arad) ,and the straight line distance heuristic hSLD(In(Arad)) is found to be 366.
- Using the **straight-line distance** heuristic **hSLD ,**the goal state can be reached faster.



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

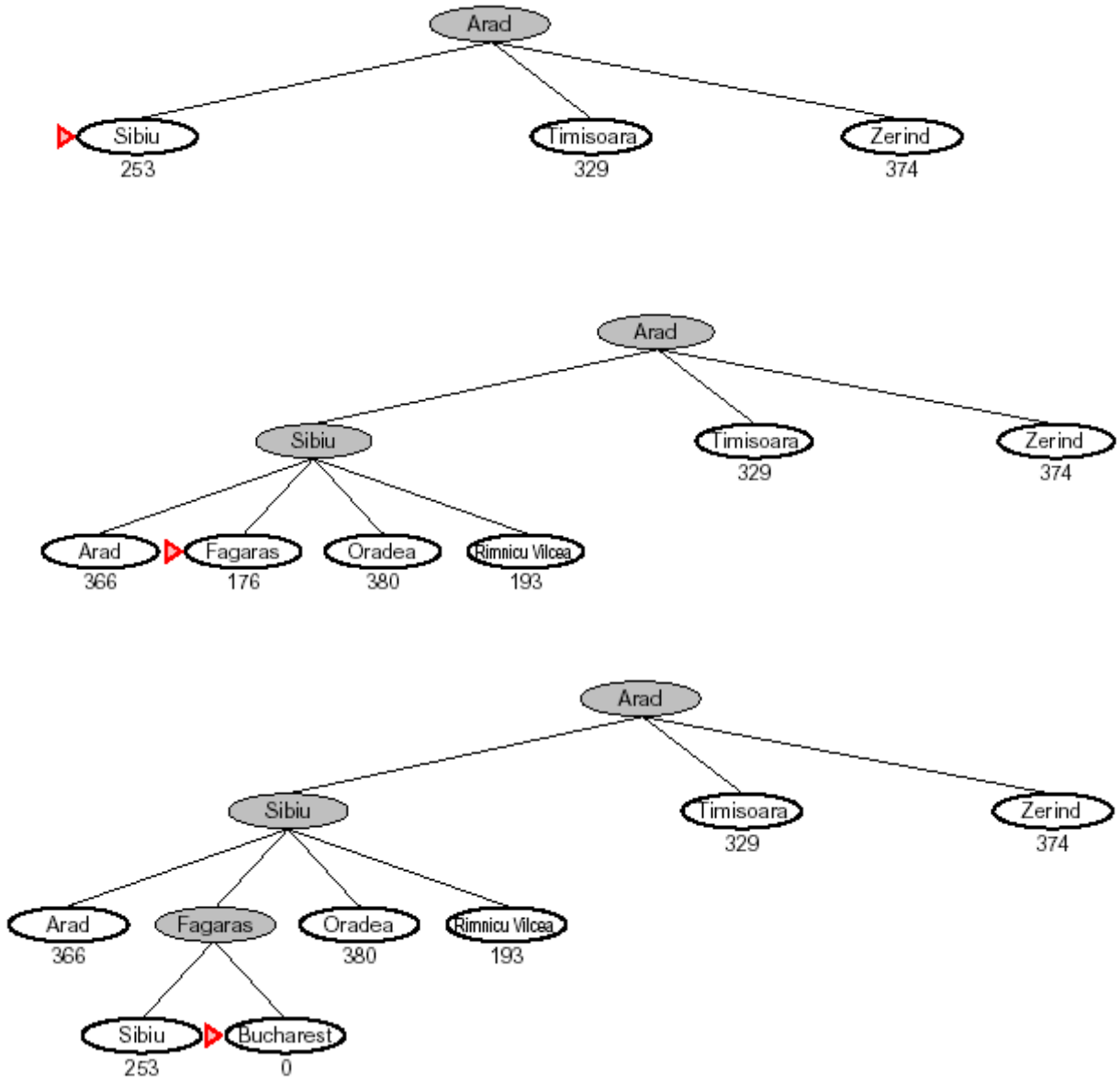


Figure 2.2 shows the progress of greedy best-first search using hSLD to find a path from Arad to Bucharest. The first node to be expanded from Arad will be Sibiu,because it is closer to Bucharest than either Zerind or Timisoara. The next node to be expanded will be Fagaras,because it is closest.

Fagaras in turn generates Bucharest,which is the goal.

Properties of greedy search

- **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(bm),where m is the maximum depth of the search space.

A* Search

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

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

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

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

A* Search is both optimal and complete. A* is optimal if h(n) is an admissible heuristic. The obvious example of admissible heuristic is the straight-line distance hSLD. It cannot be an overestimate.

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

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

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

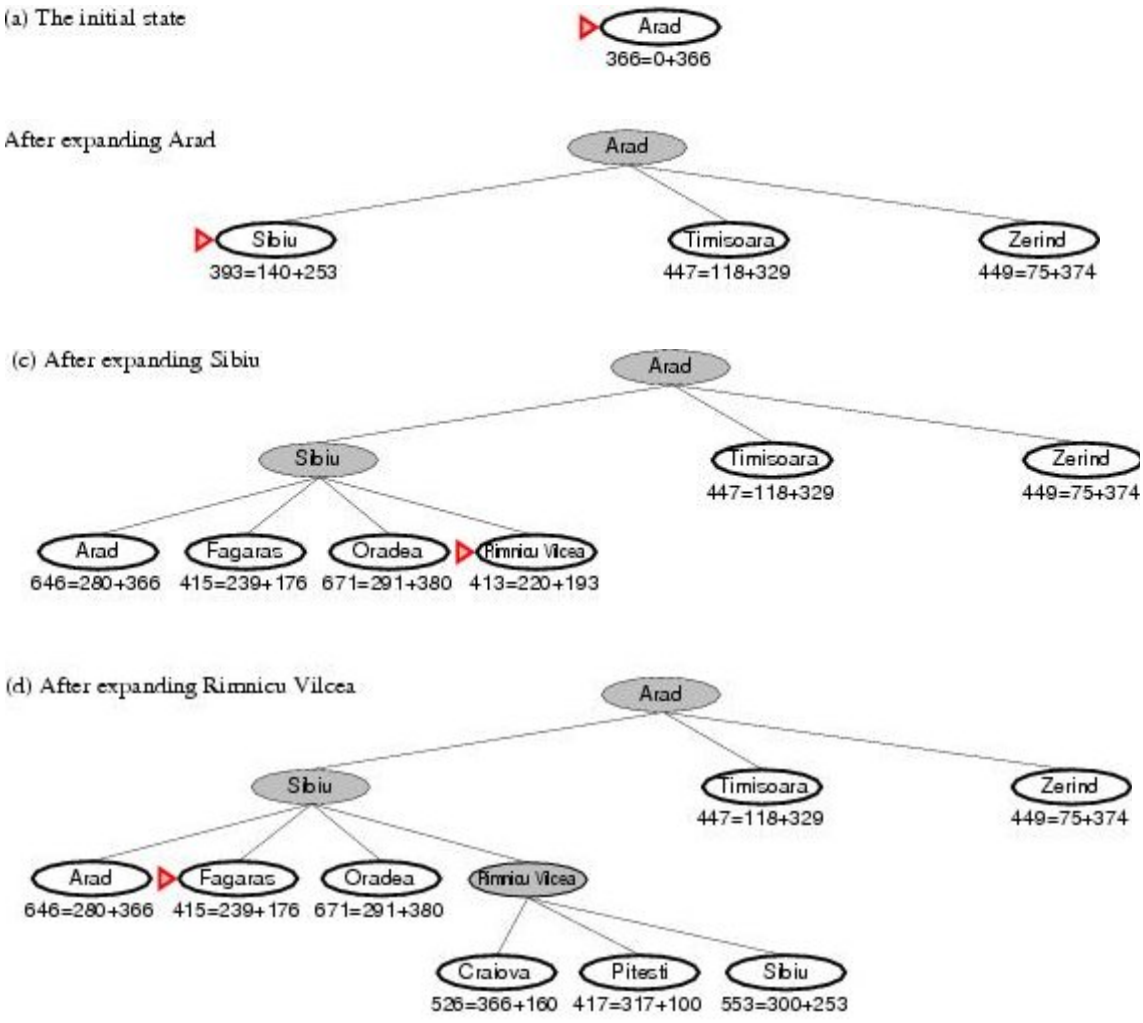
Recursive Best-first Search(RBFS) :-

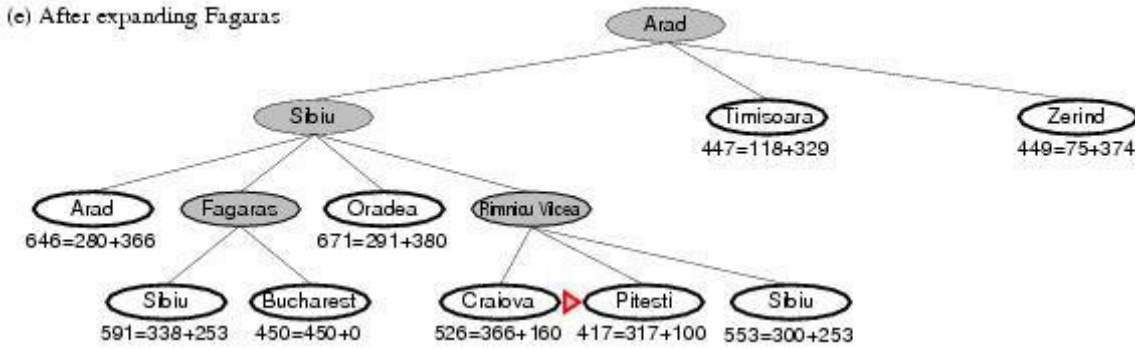
Recursive best-first search is a simple recursive algorithm that attempts to mimic the operation of standard best-first search,but using only linear space. The algorithm is shown in figure 2.4.

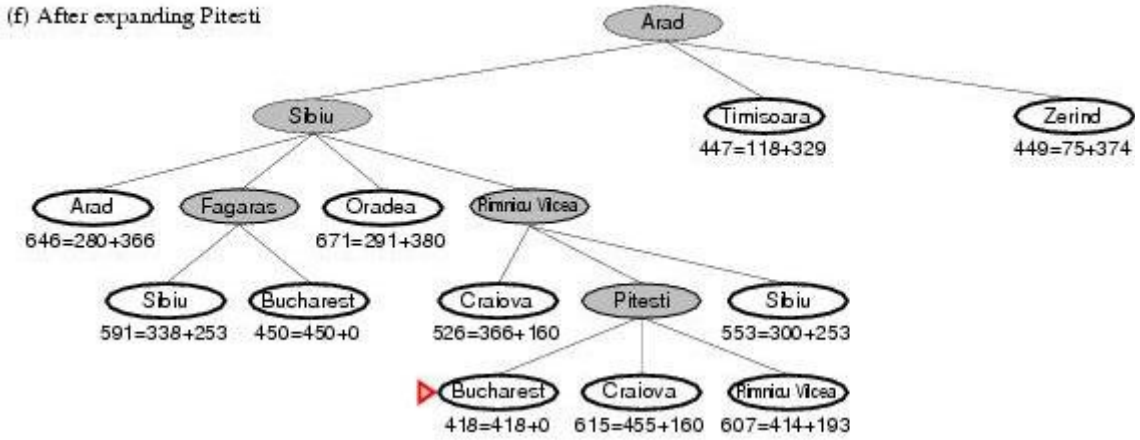
Its structure is similar to that of recursive depth-first search,but rather than continuing indefinitely down the current path,it keeps track of the f-value of the best alternative path available from any ancestor of the current node.

If the current node exceeds this limit,the recursion unwinds back to the alternative path. As the recursion unwinds,RBFS replaces the f-value of each node along the path with the best f-value of its children.

Figure 2.5 shows how RBFS reaches Bucharest.

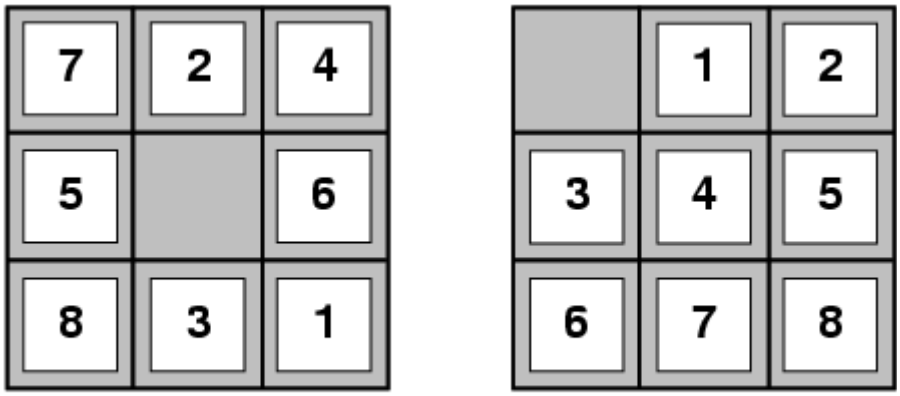






Heuristic Functions

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 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 322 approximately = 3.1 X 1010 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 1013.

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

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

(1) h1 = the number of misplaced tiles.

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

(2) h2 = the sum of the distances of the tiles from their goal positions. This is called **the city block distance** or **Manhattan distance.** h2 is admissible ,because all any move can do is move one tile one step closer to the goal.

Tiles 1 to 8 in start state give a Manhattan distance of

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

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

The Effective Branching factor :-

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

N + 1 = 1 + b* + (b*)2+…+(b*)d

A well designed heuristic would have a value of b* close to 1,allowing failru large problems to be solved.

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

The results suggest that h2 is better than h1,and is far better than using iterative deepening search.

For a solution length of 14,A* with h2 is 30,000 times more efficient than uninformed iterative deepening search.

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

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**.

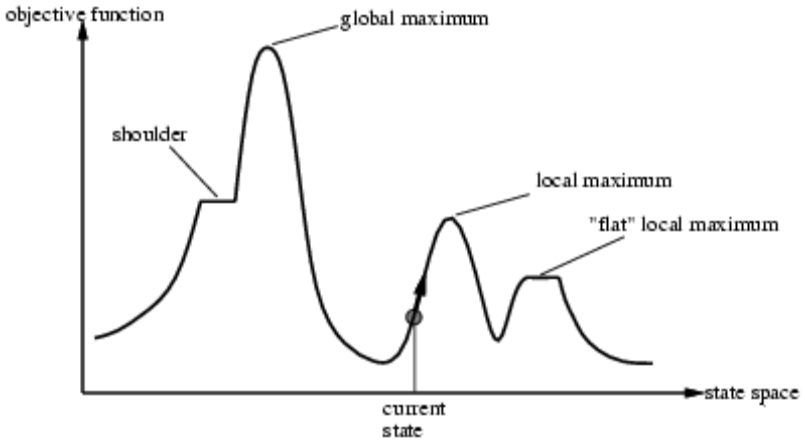
State Space Landscape

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**.



Hill-climbing search

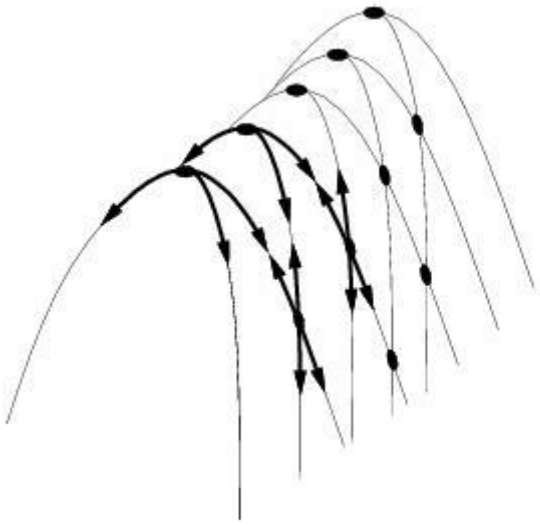
The **hill-climbing** search algorithm as shown in figure 2.9, is simply a loop that continually moves in the direction of increasing value – that is,**uphill**. It terminates when it reaches a **—peak||** where no neighbor has a higher value.

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.



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.

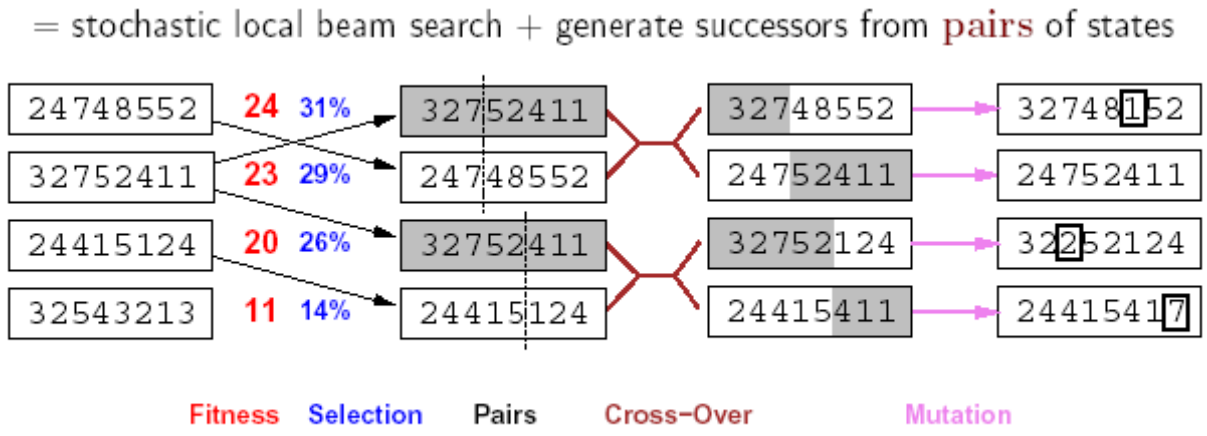
Figure 2.11 shows simulated annealing algorithm. It is quite similar to hill climbing. Instead of picking the best move,however,it picks the random move.

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 E by which the evaluation is worsened.

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 log2 8 = 24 bits.



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 functionin 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.

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

o Rampant ignorance a ripe opportunity for learning Agent learns a ―map‖ of the environment

o Outcome of each action in each state

o Local search agents improve evaluation function accuracy

o Update estimate of value at each visited state

o Would like to infer higher-level domain model

o Example: ―Up‖ in maze search increases y -coordinate Requires

o Formal way to represent and manipulate such general rules (so far, have hidden rules within the successor function)

o Algorithms that can construct general rules based on observations of the effect of actions

CONSTRAINT SATISFACTION PROBLEMS(CSP) :-

A **Constraint Satisfaction Problem**(or CSP) is defined by a set of **variables ,**X1,X2,….Xn,and a set of constraints C1,C2,…,Cm. Each variable Xi has a nonempty **domain** D,of possible **values**. Each constraint Ci 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,{Xi = vi,Xj = vj,…}. 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**.

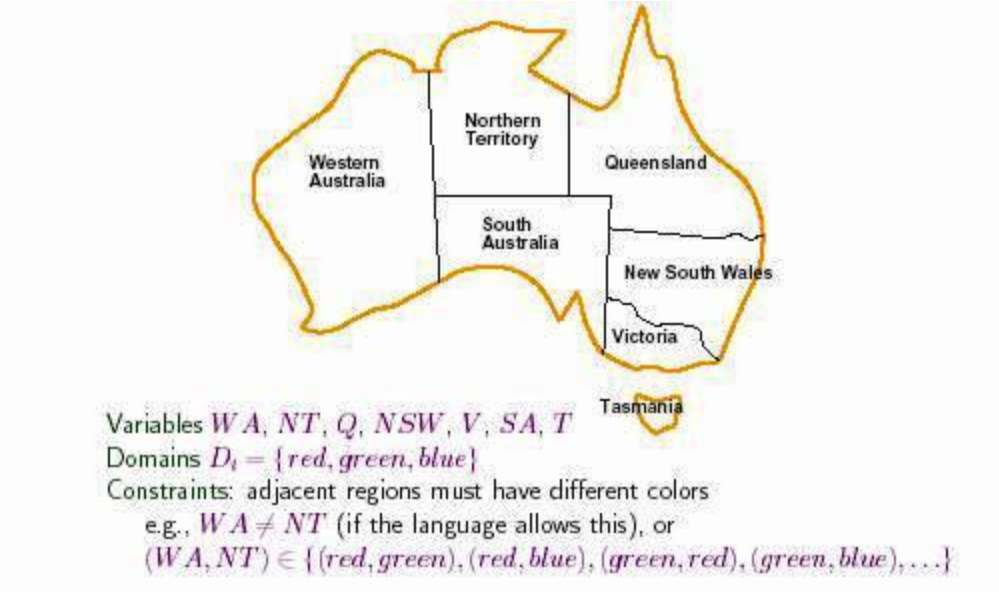
Example for Constraint Satisfaction Problem :

Figure 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}.



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 Q1,Q2,…..Q8 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(dn) – 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

Startjob1 + 5 <= Startjob3.

(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.

ADVERSARIAL SEARCH :-

Competetive 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 **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 :

o The **initial state**,which includes the board position and identifies the player to move.

o A **successor function**,which returns a list of (*move,state*) pairs,each indicating a legal move and the resulting state.

o A **terminal test**,which describes when the game is over. States where the game has ended are called **terminal states**.

o 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

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:

o 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.

o 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(bd/2) nodes to pick the best move,instead of O(bd) 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.

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.

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

o Minimax and alpha-beta pruning require too much leaf-node evaluations.

May be impractical within a reasonable amount of time.

o SHANNON (1950):

o Cut off search earlier (replace TERMINAL-TEST by CUTOFF-TEST)

o 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

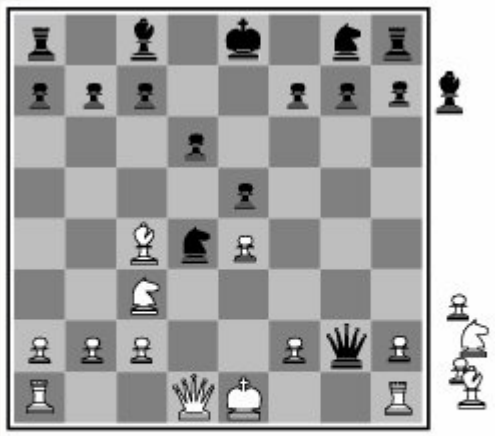
The introductory chess books give an approximate material value for each piece : each pawn is worth 1,a knight or bishop is worth 3,a rook 3,and the queen 9. These feature values are

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 :

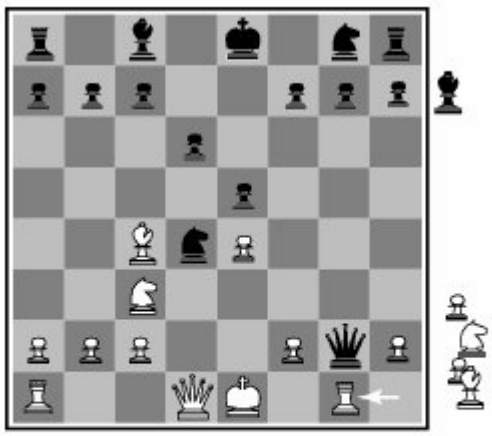
Eval(s) = w1 f1(s) + w2 f2(s) + … + wn fn(s)

• 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.