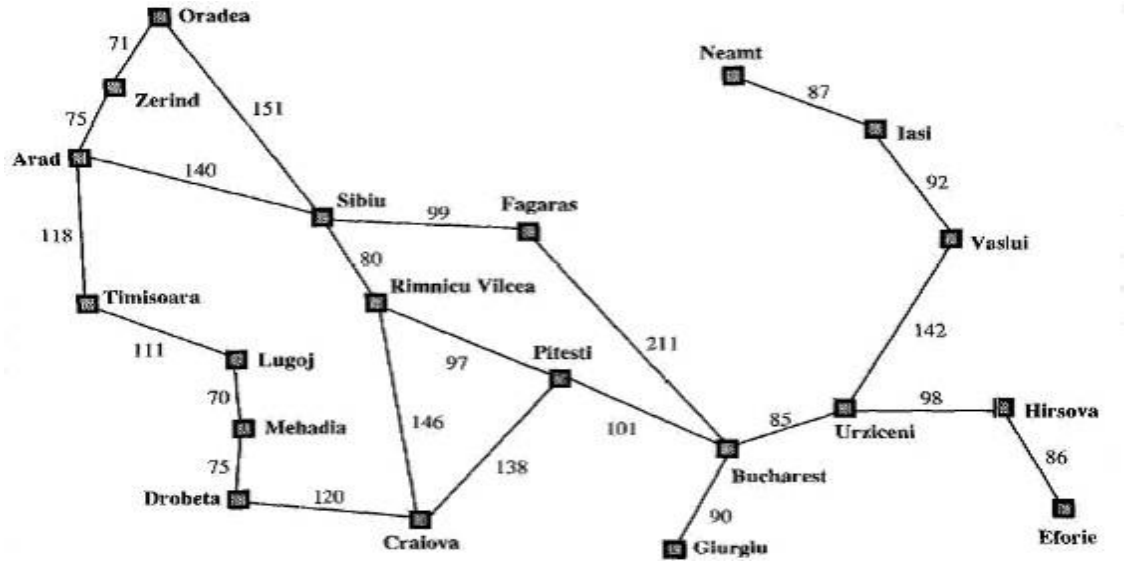
**UNIT II PROBLEM SOLVING METHODS 12**

Search Strategies: Uninformed search - Informed search - Heuristics Functions - Local Search Algorithms and Optimization Problems - Constraint Satisfaction Problems – Constraint Propagation - Backtracking Search

## 2.1 Problem solving Methods - Searching for Solutions

Search techniques use an explicit **search tree** that is generated by the initial state and the successor function that together define the state space. In general, we may have a search graph rather than a search tree, when the same state canbe reached from multiple paths

**Example Route finding problem**




The root of the search tree is a **search node** corresponding to the initial state, In(Arad).

The first step is to test whether this is a goal state.

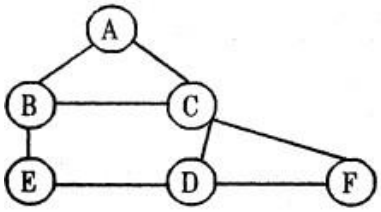
Apply the successor function to the current state, and **generate** a new set of states

In this case, we get three new states: In(Sibiu),In(Timisoara), and In(Zerind). Now we must choose which of these three possibilities to consider further. Continue choosing, testing, and expanding until either a solution is found or there are no more states to be expanded.

The choice of which state to expand is determined by the **search strategyTree Search**

**algorithm**

**Task : Find a path to reach F from A**

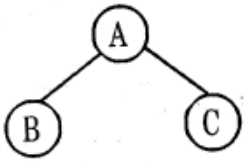



1. Start the sequence with the initial state and check whether it is a goal state ornot.

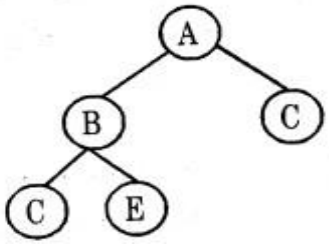
a, If it is a goal state return success.
b. Otherwise perform the following sequence of steps

From the initial state (current state) generate and expand the new set of states. The collection of nodes that have been generated but not expanded is called as fringe. Each element of the fringe is a leaf node, a node with no successors in the tree.

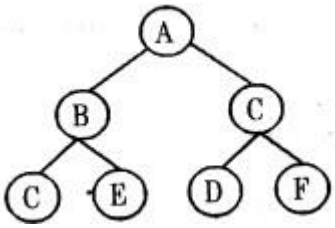
**Expanding A**




**Expanding B**




**Expanding C**



Sequence of steps to reach the goal state F from (A = A - C - F)

2. **Search strategy:** In the above example we did the sequence of choosing, testing and expanding until a solution is found or until there are no more states to be expanded. The choice of which state to expand first is determined by search strategy.
3. **Search tree:** The tree which is constructed for the search process over the state space.
4. **Search node:** The root of the search tree that is the initial state of the problem.

**The general tree search algorithm**

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

There are many ways to represent nodes, but we will assume that a node is adata structure with five components:

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

The collection of nodes represented in the search tree is defined using set orqueue representation.

**Set :** The search strategy would be a function that selects the next node to beexpanded from the set

**Queue:** Collection of nodes are represented, using queue. The queue operationsare defined as:

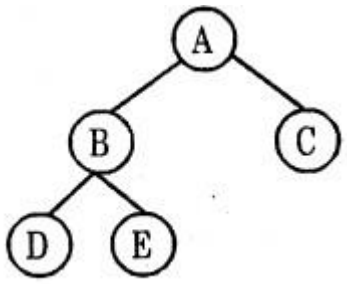
**MAKE-QUEUE(elements)** - creates a queue with the given elements **EMPTY(queue)**-returns true only if there are no more elements in the queue. **REM0VE-FIRST(queue)** - removes the element at the front of the queue andreturns it
**INSERT ALL (elements, queue)** - inserts set of elements into the queue andreturns the resulting queue.
**FIRST (queue)** - returns the first element of the queue.
**INSERT (element, queue)** - inserts an element into the queue and returns theresulting queue

The general tree search algorithm with queue representation

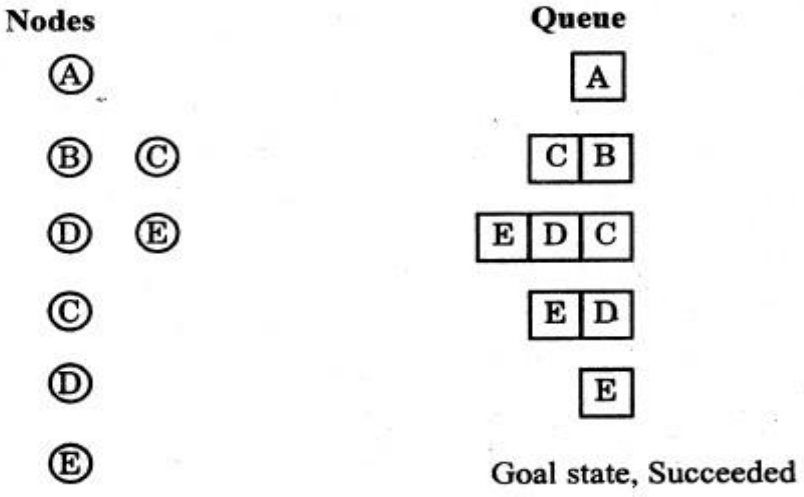
```
function TREE-SEARCH(problem,fringe) returns a
solution, or failure
fringe  <-  INSERT(MAKE-NODE(INITIAL-STATE[problem]),
fringe)
loop do
if EMPTY?(fringe) then return failure
node <- REMOVE-FIRST(fringe)
ifGOAL-TEST[problenl]applied to STATE[node] succeeds
then return SOLUTION(node)
fringe <- INSERT-ALL(EXPAND(node, problem),fringe)
```

```
function EXPAND(node, problem) returns a set of nodes
successors <- the empty set
for each <action, result> in SUCCESSOR-FN
[problem](STATE[node])do
S <- a new NODE
STATE[s] <- result
PARENT-NODE[s] <- node
ACTION[s] <- action
PATH-COST[s]  <-  PATH-COST[node]+STEP-COST(node,action,s)
DEPTH[s] <- DEPTH[node] + 1
add s to successors
return successors
```

Example: Route finding problem




Task. : Find a path to reach E using Queuing function in general tree searchalgorithm

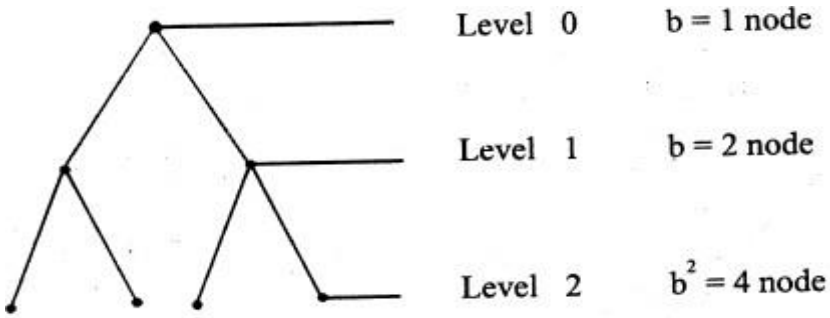



Measuring problem solving performance

The search strategy algorithms are evaluated depends on four importantcriteria's. They are:

**(i) Completeness :** The strategy guaranteed to find a solution when there isone.
**(ii) Time complexity :** Time taken to run a solution
**(iii) Space complexity :** Memory needed to perform the search.
**(iv) Optimality :** If more than one way exists to derive the solution then thebest one is Selected

**Definition of branching factor (b):** The number of nodes which is connectedto each of the node in the search tree. Branching factor is used to find space and time complexity of the search strategy




**2.2 Search Strategies -Solving Problems by Searching**

The searching algorithms are divided into two categories

1. **Uninformed Search Algorithms (Blind Search)**
2. **Informed Search Algorithms (Heuristic Search)**

There are six Uninformed Search Algorithms

1. **Breadth First Search**
2. **Uniform-cost search**
3. **Depth-first search**
4. **Depth-limited search**
5. **Iterative deepening depth-first search**
6. **Bidirectional Search**

There are three Informed Search Algorithms

1. **Best First Search**
2. **Greedy Search**
3. **A* Search**

**Blind search Vs Heuristic search**

| Blind search | Heuristic search |
|---|---|
| No information about the number of steps (or) path cost from current state to goal state | The path cost from the current state to the goal state is calculated, to select the minimum path cost as the next state. |
| Less effective in search method | More effective in search method |
| Problem to be solved with the giveninformation | Additional information can be added asassumption to solve the problem |

**2.3 Uninformed Search Algorithms (Blind Search)**

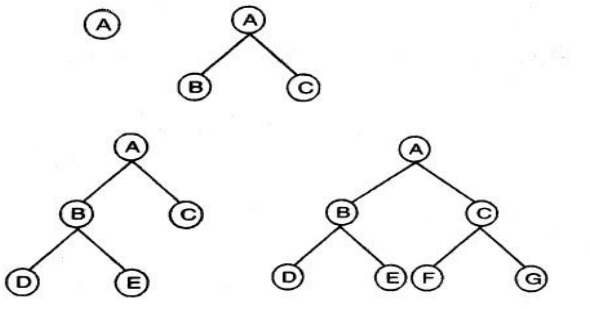
**2.3.1 Breadth-first search**

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

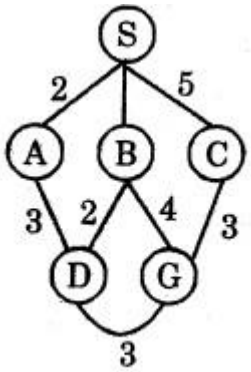
Breadth-first search can be implemented by calling TREE-SEARCH with an emptyfringe that is a first-in-first-out (FIFO) queue, assuring that the nodes that are visited first will be expanded first.

In other words, calling **TREE-SEARCH(Problem, FIFO-QUEUE())**results in a breadth-first search. The FIFO queue puts all newly generated successors at the end of the queue, which means that shallow nodes are expanded before deeper nodes

**Breadth first search trees after node expansions**



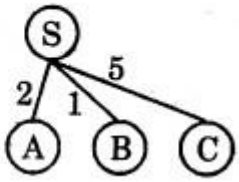

**Example: Route finding problem**



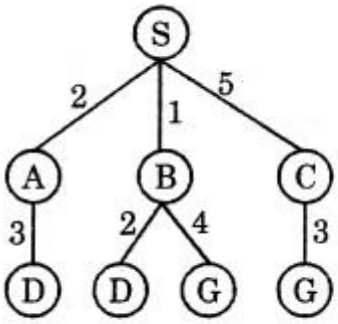

**Task: Find a ,path from. S to G using BFS**

(i)

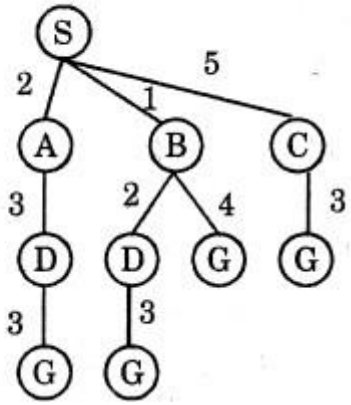

(ii)




(iii)




(iv)




The path in the 2nd depth level is selected, (i.e) SBG{or) SCG.

**Algorithm :**

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
node ←a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
frontier ←a FIFO queue with node as the only element
explored ←an empty set
loop do
if EMPTY?( frontier ) then return failure
node←POP( frontier ) /* chooses the shallowest node in frontier */
add node.STATE to explored
for each action in problem.ACTIONS(node.STATE) do
child ←CHILD-NODE(problem, node, action)
if child.STATE is not in explored or frontier then
if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
frontier ←INSERT(child, frontier )
```

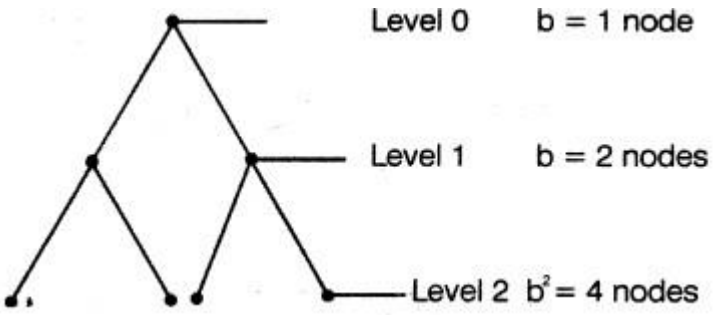
**Time and space complexity:**

**Example:**

**Time complexity**

$= 1 + b + b^2 + \ldots\ldots\ldots\ldots + b^d$

$= \mathbf{O(b^d)}$




The **space complexity** is same as time complexity because all the leaf nodes ofthe tree must be maintained in memory at the same time = $\mathbf{O(b^d)}$

**Completeness:** Yes

**Optimality**: Yes, provided the path cost is a non decreasing function of thedepth of the node

**Advantage:** Guaranteed to find the single solution at the shallowest depth level

**Disadvantage:** Suitable for only smallest instances problem (i.e.) (number oflevels to be minimum (or) branching factor to be minimum)
')

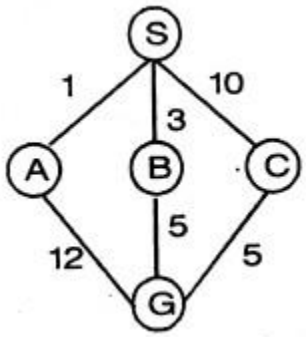
**2.3.2 Uniform-cost search**

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
node ←a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
frontier ←a priority queue ordered by PATH-COST, with node as the only
element
explored ←an empty set
loop do
if EMPTY?( frontier ) then return failure
node←POP( frontier ) /* chooses the lowest-cost node in frontier */
if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
add node.STATE to explored
for each action in problem.ACTIONS(node.STATE) do
child ←CHILD-NODE(problem, node, action)
if child.STATE is not in explored or frontier then
frontier ←INSERT(child, frontier )
else if child.STATE is in frontier with higher PATH-COST then
replace that frontier node with child
```

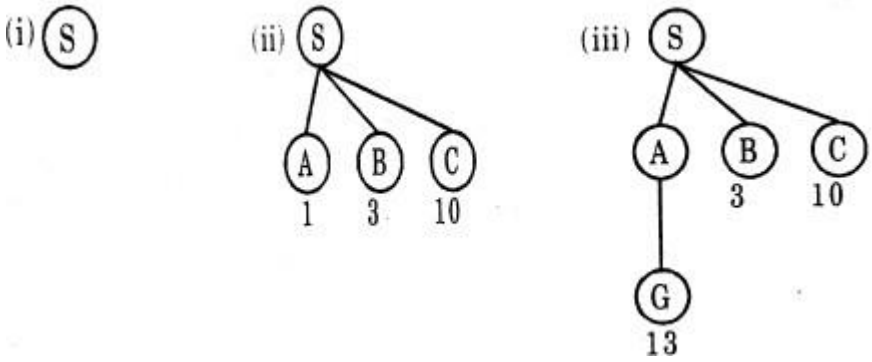
Breadth-first search is optimal when all step costs are equal, because it always expands the shallowest unexpanded node. By a simple extension, we can find an algorithm that is optimal with any step cost function. Instead of expanding the shallowest node, **uniform-cost search** expands the node n with the lowest path cost. Note that if all step costs are equal, this is identical to breadth-first search.

Uniform-cost search does not care about the number of steps a path has, butonly about their total cost.

**Example: Route finding problem**

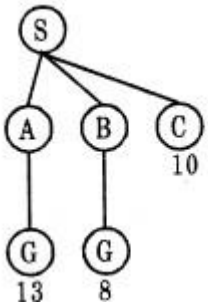



**Task : Find a minimum path cost from S to G**




Since the value of A is less it is expanded first, but it is not optimal.

**B to be expanded next**




SBG is the path with minimum path cost.

No need to expand the next path SC, because its path cost is high to reach Cfrom S, as well as goal state is reached in the previous path with minimum cost.

**Time and space complexity:**

Time complexity is same as breadth first search because instead of depth levelthe minimum path cost is considered.

**Time complexity**: **O(b $^{d}$) Space complexity**: **O(b $^{d}$)Completeness**: Yes **Optimality:** Yes

**Advantage:** Guaranteed to find the single solution at minimum path cost.

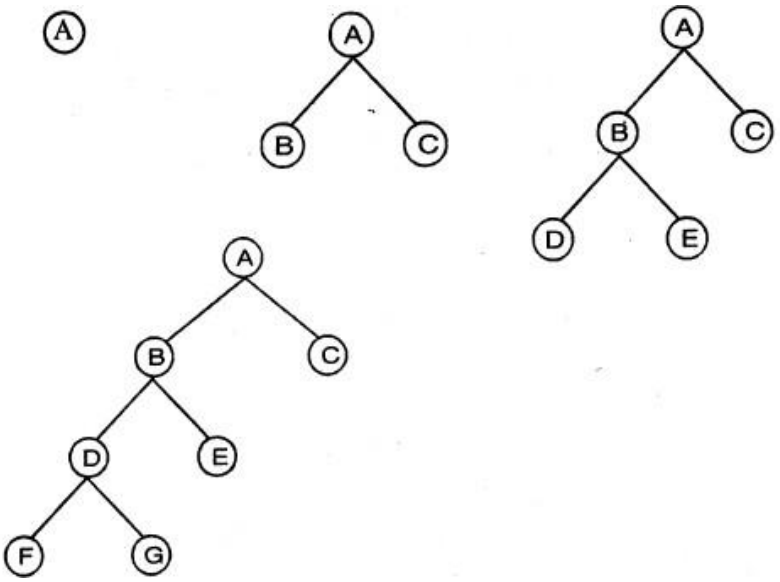
**Disadvantage:** Suitable for only smallest instances problem.

## 2.3.3 Depth-first search

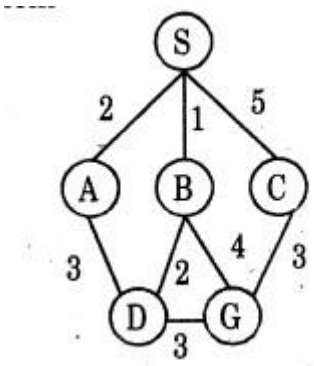
**Depth-first search** always expands the deepest node in the current fringe of the search tree

The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As those nodes are expanded, they are dropped from the fringe, so then the search "backs up" to the next shallowest node that still has unexplored successors. This strategy can be implemented by TREE- SEARCH with a last-in-first-out (LIFO) queue, also known as a stack.

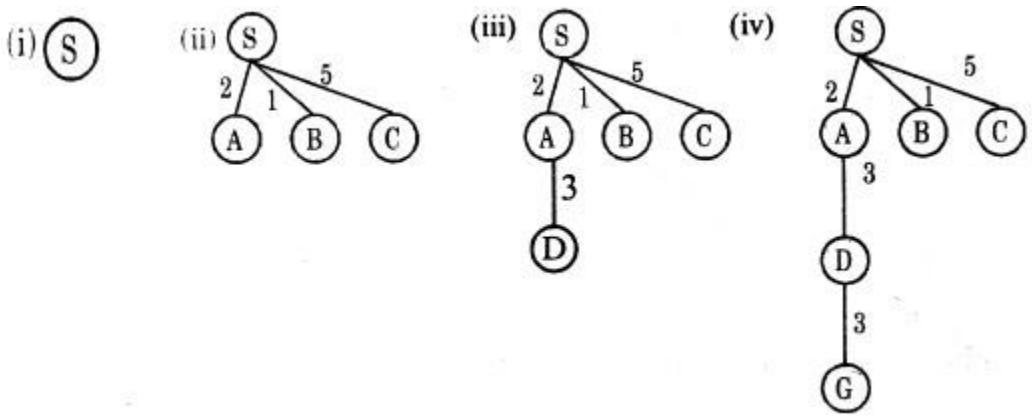
Depth first search tree with 3 level expansion




**Example: Route finding problem**




**Task: Find a path from S to G using DFS**




The path in the 3rd depth level is selected. (i.e. S-A-D-G

**Algorithm:**

```
function DFS(problem) return a solution or failure
TREE-SEARCH(problem,  LIFO-QUEUE())
```

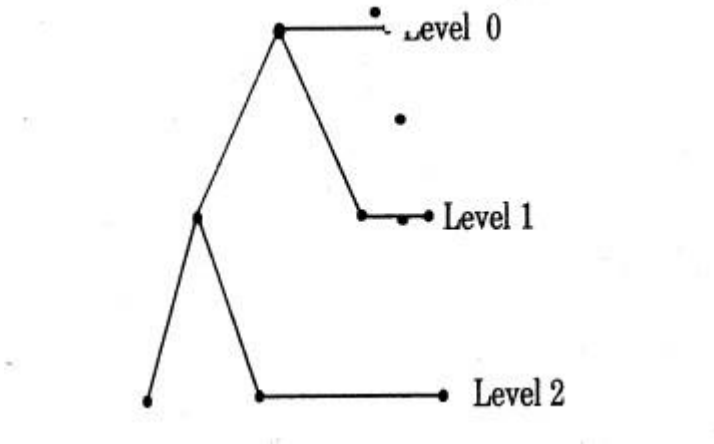
**Time and space complexity:**

In the worst case depth first search has to expand all the nodes

**Time complexity : $O(b^m)$.**

The nodes are expanded towards one particular direction requires memory foronly that nodes.

**Space complexity : O(bm)**




b=2
m = 2 :. bm=4 **Completeness:** No**Optimality:** No

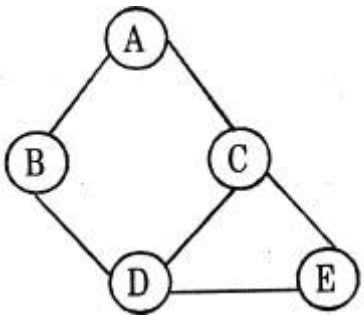
**Advantage:** If more than one solution exists (or) number of levels is high then DFS is best because exploration is done only in a small portion of the whole space.

**Disadvantage:** Not guaranteed to find a solution

### 2.3.4 Depth - limited search

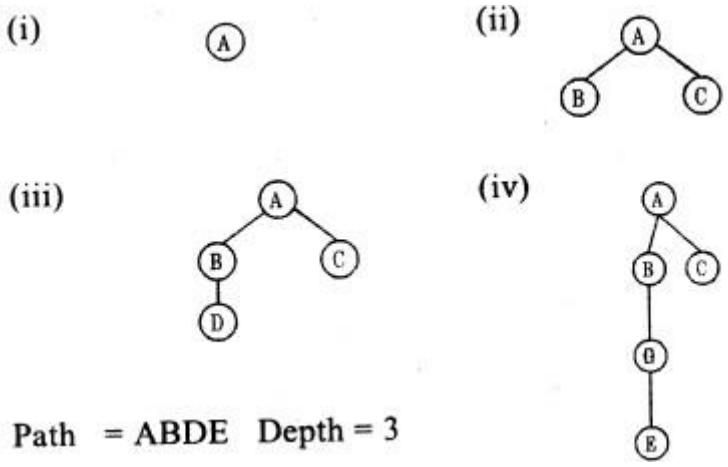
1. Definition: A cut off (maximum level of the depth) is introduced in this search technique to overcome the disadvantage of depth first search. The cutoff value depends on the number of states.

Example: Route finding problem




The number of states in the given map is 5. So, it is possible to get the goalstate at a maximum depth of 4. Therefore the cutoff value is 4

**Task : Find a path from A to E.**




**A recursive implementation of depth-limited search**

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a
solution, or failure/cutoff
return  RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE  [problem]),
problem, limit)
```

```
function RECURSIVE-DLS(node, problem, limit) returns a
solution, or failure/cutoff
cutoff-occurred? <- false
if GOAL-TEST[problem](STATE[node]) then return
SOLUTION(node)
else if DEPTH[node] =limit then return cutoff
else for each successor in EXPAND(node, problem) do
result <- RECURSIVE-DLS(successor, problem, limit)
if result = cutoff then cutoff-occurred?<- true
else if result ≠ failure then return result
if cutoff-occurred? then return cutoff else return
failure
```

**Time and space complexity:**

The worst case time complexity is equivalent to BFS and worst case DFS.

**Time complexity : $O(b^l)$**

The nodes which is expanded in one particular direction above to be stored.

**Space complexity : O(bl)**

**Optimality**: No, because not guaranteed to find the shortest solution first in thesearch technique.

**Completeness :** Yes, guaranteed to find the solution if it exists.

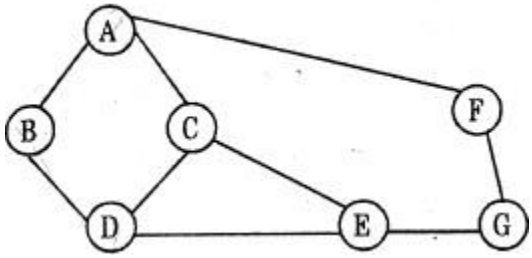
**Advantage**: Cut off level is introduced in the DFS technique

**Disadvantage :** Not guaranteed to find the optimal solution.Iterative deepening search

## 2.3.5 Iterative deepening search

Definition: Iterative deepening search is a strategy that sidesteps the issue ofchoosing the best depth limit by trying all possible depth limits.

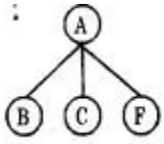
**Example: Route finding problem**



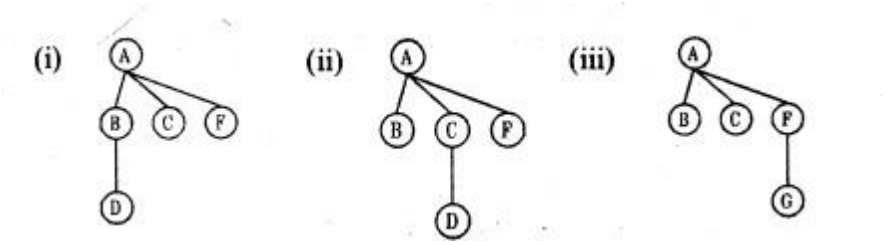

**Task: Find a path from A to G**

Limit = 0


Limit = 1



Limit = 2




Solution: The goal state G can be reached from A in four ways. They are:

1. **A – B – D - E – G --------- - Limit 42. A - B - D - E - G Limit 4**
3. **A - C - E - G ----------Limit 3**
4. **A - F - G ---------Limit2**

Since it is a iterative deepening search it selects lowest depth limit (i.e.) A-F-G isselected as the solution path.

The iterative deepening search algorithm :

```
function ITERATIVE-DEEPENING-SEARCH (problem) returns a
solution, or failure
inputs : problem
for depth <- 0 to ∞ do
result <-DEPTH-LIMITED-SEARCH(problem, depth)
if result ≠cutoff then return result
```

**Time and space complexity :**

Iterative deepening combines the advantage of breadth first search and depth first search (i.e) expansion of states is done as BFS and memory requirement is equivalent to DFS.

**Time complexity : $O(b^d)$ Space Complexity : O(bd)**

**Optimality:** Yes, because the order of expansion of states is similar to breadth first search.

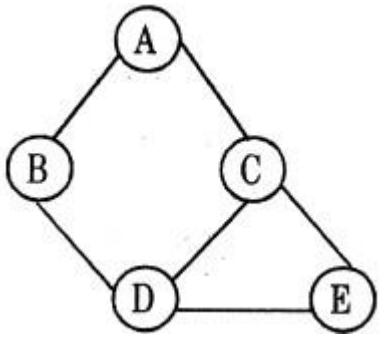
**Completeness:** yes, guaranteed to find the solution if it exists.
**Advantage:** This method is preferred for large state space and the depth of thesearch is not known.

**Disadvantage :** Many states are expanded multiple times
**Example :** The state D is expanded twice in limit 2

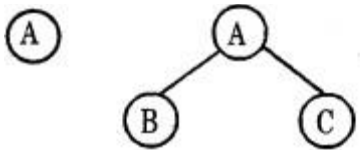
## 2.3.6 Bidirectional search

**Definition :** Bidirectional search is a strategy that simultaneously search both the directions (i.e.) forward from the initial state and backward from the goal, and stops when the two searches meet in the middle.

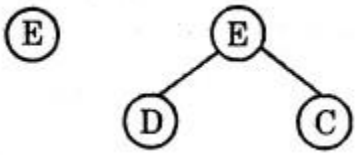
**Example: Route finding problem**




**Task : Find a path from A to E.Search from forward (A) :**




**Search from backward (E) :**




**Time and space complexity:**

The forward and backward searches done at the same time will lead to thesolution in $\mathbf{O(2b^{d/2}) = O(b^{d/2})}$step, because search is done to go only halfway

If the two searches meet at all, the nodes of at least one of them must all beretained in memory requires **$O(b^{d/2})$** space.

**Optimality:** Yes, because the order of expansion of states is done in both thedirections.

**Completeness:** Yes, guaranteed to find the solution if it exists.

**Advantage :** Time and space complexity is reduced.

**Disadvantage:** If two searches (forward, backward) does not meet at all, complexity arises in the search technique. In backward search calculating predecessor is difficult task. If more than one goal state 'exists then explicit, multiple state search is required

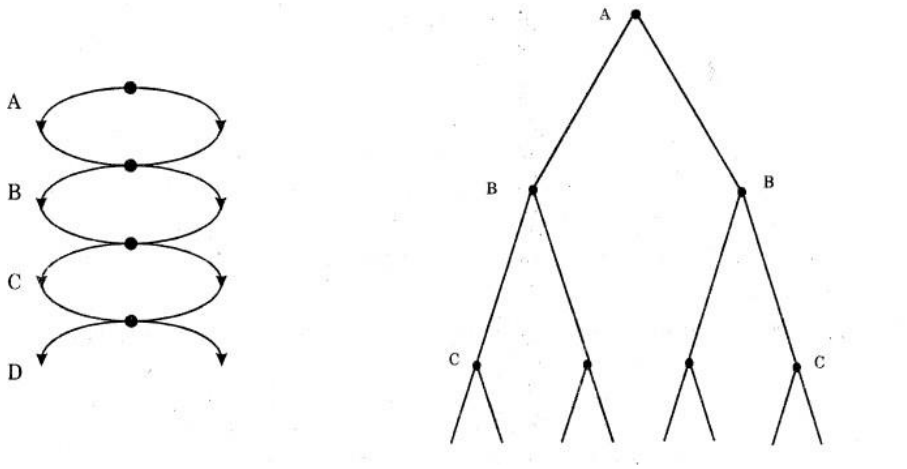
**Comparing uninformed search strategies**

| **Criterion** | **Breadth First** | **Uniform Cost** | **Depth First** | **Depth Limited** | **Iterative Deepening** | **Bi direction** |
|---|---|---|---|---|---|---|
| **Complete** | **Yes** | **Yes** | **No** | **No** | **Yes** | **Yes** |
| **Time** | **$O(b^d)$** | **$O(b^d)$** | **$O(b^m)$** | **$O(b^l)$** | **$O(b^d)$** | **$O(b^{d/2})$** |
| **Space** | **$O(b^d)$** | **$O(b^d)$** | **$O(bm)$** | **O(bl)** | **O(bd)** | **$O(b^{d/2})$** |
| **Optimal** | **Yes** | **Yes** | **No** | **No** | **Yes** | **Yes** |

**Avoiding Repeated States**

The most important complication of search strategy is expanding states thathave already been encountered and expanded before on some other path

**A state space and its exponentially larger search tree**




The repeated states can be avoided using three different ways. They are:

1. Do not return to the state you just came from (i.e) avoid any successor that isthe same state as the node's parent.
2. Do not create path with cycles (i.e) avoid any successor of a node that is thesame as any of the node's ancestors.
3. Do not generate any state that was ever generated before.

The general TREE-SEARCH algorithm is modified with additional data structure,such as :

Closed list - which stores every expanded node.Open list - fringe of unexpanded nodes.
If the current node matches a node on the closed list, then it is discarded and it is not considered for expansion. This is done with GRAPH-SEARCH algorithm. This algorithm is efficient for problems with many repeated states

```
function GRAPH-SEARCH (problem, fringe) returns a
solution, or failure
closed <- an empty set
fringe <- INSERT (MAKE-NODE(INITIAL-STATE[problem]),
fringe)
loop do
if EMPTv?(fringe) then return failure
node <- REMOVE-FIKST (fringe)
if GOAL-TEST [problem ](STATE[node]) then return SOLUTION
(node)
if STATE [node] is not in closed then
add STATE [node] to closed
fringe <- INSERT-ALL(EXPAND(node, problem), fringe)
```

The worst-case time and space requirements are proportional to the size of thestate space, this may be much smaller than $O(b^d)$

## 2.4 Informed Heuristic Search Strategies

An **informed search** strategy uses problem-specific knowledge beyond the definition of the problem itself and it can find solutions more efficiently than an uninformed strategy.

The general approach is best first search that uses an evaluation function in **TREE-SEARCH or GRAPH-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 node with the lowest evaluation is selected for expansion, because the evaluation measures distance to the goal.

Best-first search can be implemented within our general search framework via a priority queue, a data structure that will maintain the fringe in ascending order off –values

### Implementation of Best-first search using general search algorithm

```
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 TREE-SEARCH(problem, QUEUEING-FN)
```

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

**The key** component of these algorithms is a heuristic functions denoted h(n).h(n) = estimated cost of the cheapest path from node n to a goal node.

**One constraint:** if n is a goal node, then h(n) = 0

**The two types of evaluation functions are:**

**(i)** Expand the node closest to the goal state using estimated cost as the evaluation is called **greedy best first search.**

**(ii)** Expand the node on the least cost solution path using estimated cost andactual cost as the evaluation function is called **A*search**

**2.4.1 Greedy best first search (Minimize estimated cost to reach a goal)**

**Definition :** A best first search that uses h(n) to select next node to expand iscalled greedy search
**Evaluation function :** The estimated cost to reach the goal state, denoted bythe letter h(n)

```
h(n)= estimated cost of the cheapest path from the state
at node n to a goal state
```

Algorithm :

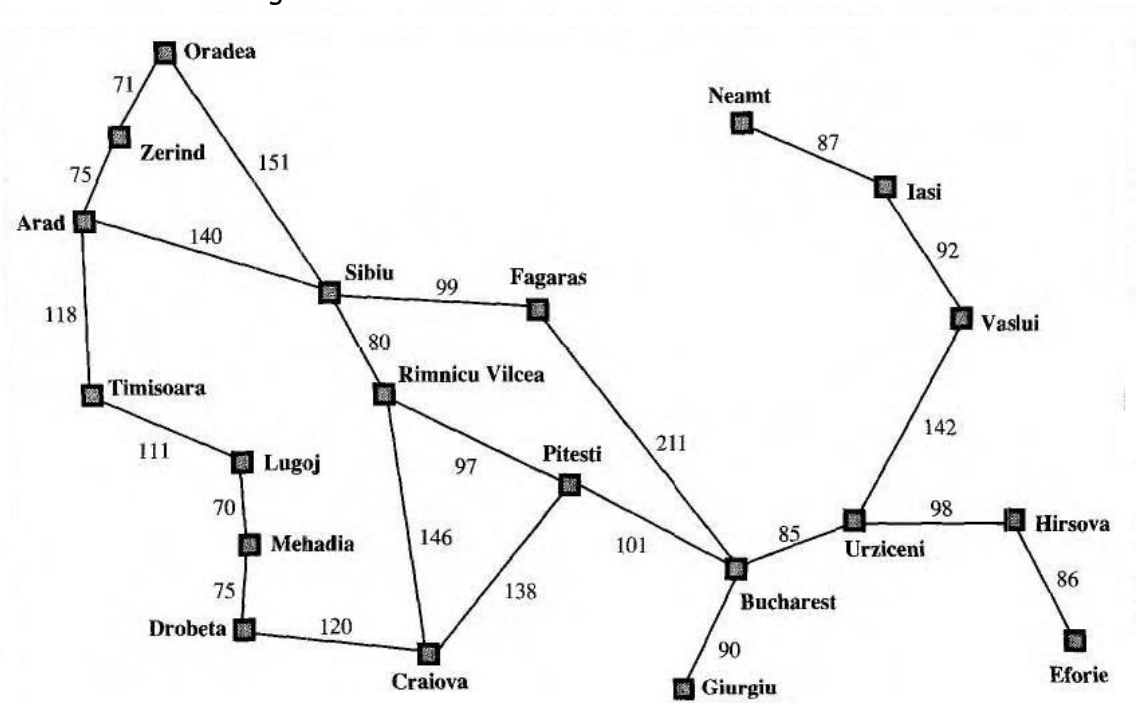
**Implementation of Best-first search using general search algorithm**

```
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 TREE-SEARCH(problem, QUEUEING-FN)
```

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

```
Function GREEDY-BEST-FIRST SEARCH (problem) returns a
solution or failure
return BEST-FIRST-SEARCH (problem, h)
```

**Example 1 : Route Finding Problem**




**Problem : Route finding Problem from Arad to Burcharest**

**Heuristic function :** A good heuristic function for route-finding problems isStraight-Line Distance to the goal and it is denoted as hSLD(n).

$h_{SLD}(n)$ = Straight-Line distance between n and the goal locatation

**Note :** The values of hSLD(n) cannot be computed from the problem descriptionitself. Moreover, it takes a certain amount of experience

**Values of hSLD-straight-line distances to Bucharest**

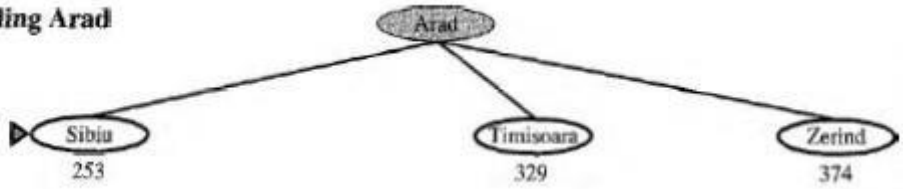
| | | | |
|---|---|---|---|
| **Arad** | 366 | **Mehadia** | 241 |
| **Bucharest** | 0 | **Neamt** | 234 |
| **Craiova** | 160 | **Oradea** | 380 |
| **Drobeta** | 242 | **Pitesti** | 100 |
| **Eforie** | 161 | **Rimnicu Vilcea** | 193 |
| **Fagaras** | 176 | **Sibiu** | 253 |
| **Giurgiu** | 77 | **Timisoara** | 329 |
| **Hirsova** | 151 | **Urziceni** | 80 |
| **Iasi** | 226 | **Vaslui** | 199 |
| **Lugoj** | 244 | **Zerind** | 374 |

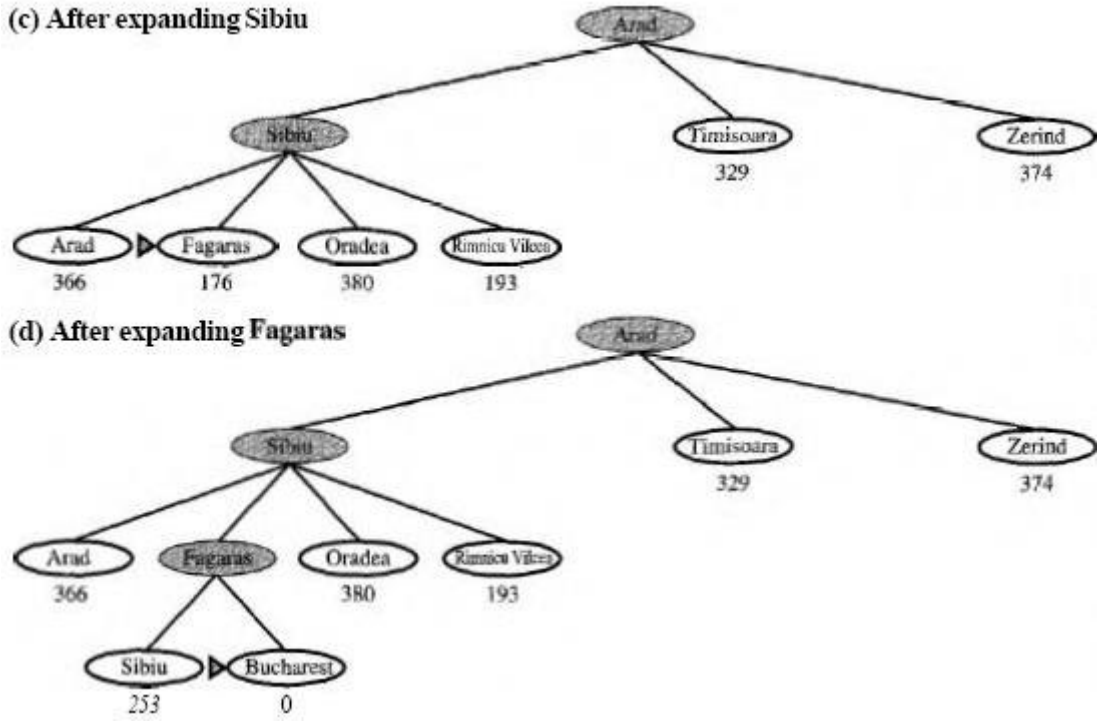
**Solution :**

From the given graph and estimated cost, the goal state is identified as B u c ha r e s t from Arad. Apply the evaluation function h (n) to find a path from Aradto Burcharest from A to B

**(a) The initial state**


**(b) After expanding Arad**






The first node to be expanded from Arad will be Sibiu, because it is closer toBucharest 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.

For this particular problem, greedy best-first search using hSLD finds a solution without ever expanding a node that is not on the solution path; hence, its search cost is minimal. It is not optimal, however: the path via Sibiu and Fagaras to Bucharest is 32 kilometers longer than the path through Rimnicu Vilcea and Pitesti. This shows why the algorithm is called "greedy'-at each step it tries to getas close to the goal as it can.

Minimizing h(n) is susceptible to false starts. Consider the problem of getting from Iasi to Fagaras. The heuristic suggests that Neamt be expanded first, because it is closest to Fagaras, but it is a dead end. The solution is to go first to Vaslui-a step that is actually farther from the goal according to the heuristic-and then to continue to Urziceni, Bucharest, and Fagaras.

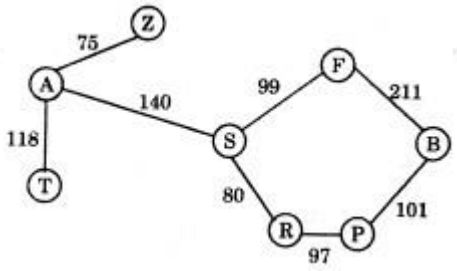
**Time and space complexity :** Greedy search resembles depth first search, since it follows one path to the goal state, backtracking occurs when it finds a dead end. The worst case time complexity is equivalent to depth first search, that is $O(b^m)$, where m is the maximum depth of the search space. The greedy

search retains all nodes in memory, therefore the space complexity is also $O(b^m)$The time and space complexity can be reduced with good heuristic function.

**Optimality :** It is not optimal, because the next level node for expansion isselected only depends on the estimated cost and not the actual cost.

**Completeness :** No, because it can start down with an infinite path and neverreturn to try other possibilities.

**Example 2 : Finding the path from one node to another node**



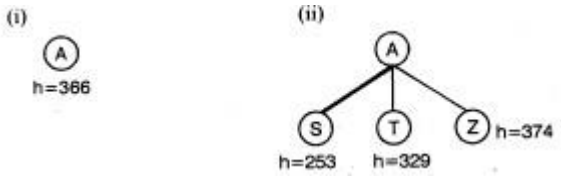

***Straight - line distance to B from A:***

A - 366
B - 0
F - 178
P - 98
R - 193
S - 253
T - 329
Z - 374

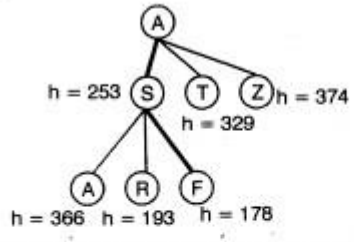
**Solution :**

From the given graph and estimated cost, the goal state IS identified as B fromA.

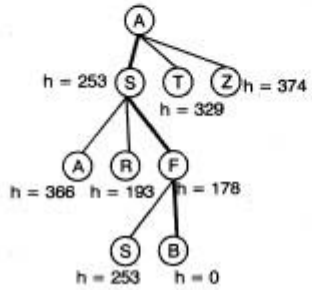
Apply the evaluation function h(n) to find a path from A to B



(iii) S is selected for next level of expansion, since *h(n)* is minimum from S, when comparing to T and Z




(iv) F is selected for next level of expansion, since *h(n)* is minimum from F.




From F, goal state B is reached. Therefore the path from A to Busing greedysearch is A - S - F - B = 450 (i.e) (140 + 99 + 211)

## 2.4.2 A* search ( Minimizing the total estimated solution cost)

The most widely-known form of best-first search is called **A*** search (pronounced"A-star search"). **A*** search is both complete and optimal.
It evaluates nodes by combining g(n), the cost to reach the node, and h(n.),thecost to get from the node to the goal

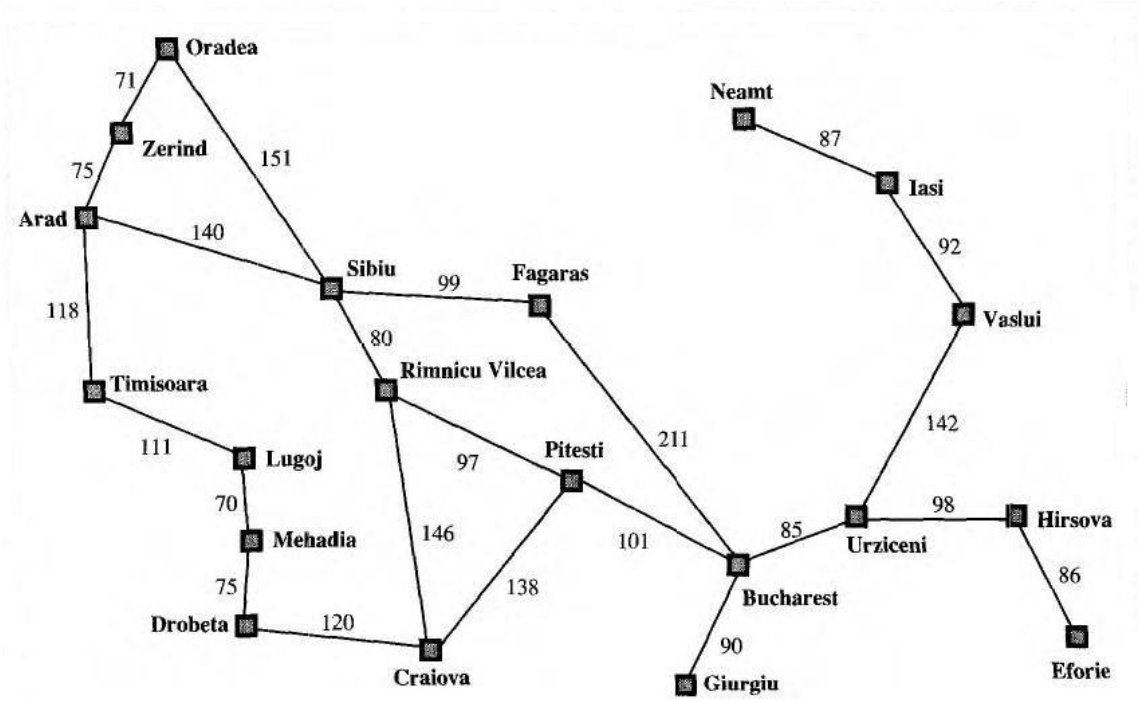
```
f(n) =g(n) + h(n)
```

g(n) - path cost from the start node to node n
h(n) - estimated cost of the cheapest path from n to the goalf (n) - estimated cost of the cheapest solution through n

**A* Algorithm**

```
function A* SEARCH(problem) returns a solution or failure
     return BEST-FIRST-SEARCH (problem, g+h)
```

**Example 1 : Route Finding Problem**




**Problem : Route finding Problem from Arad to Burcharest**

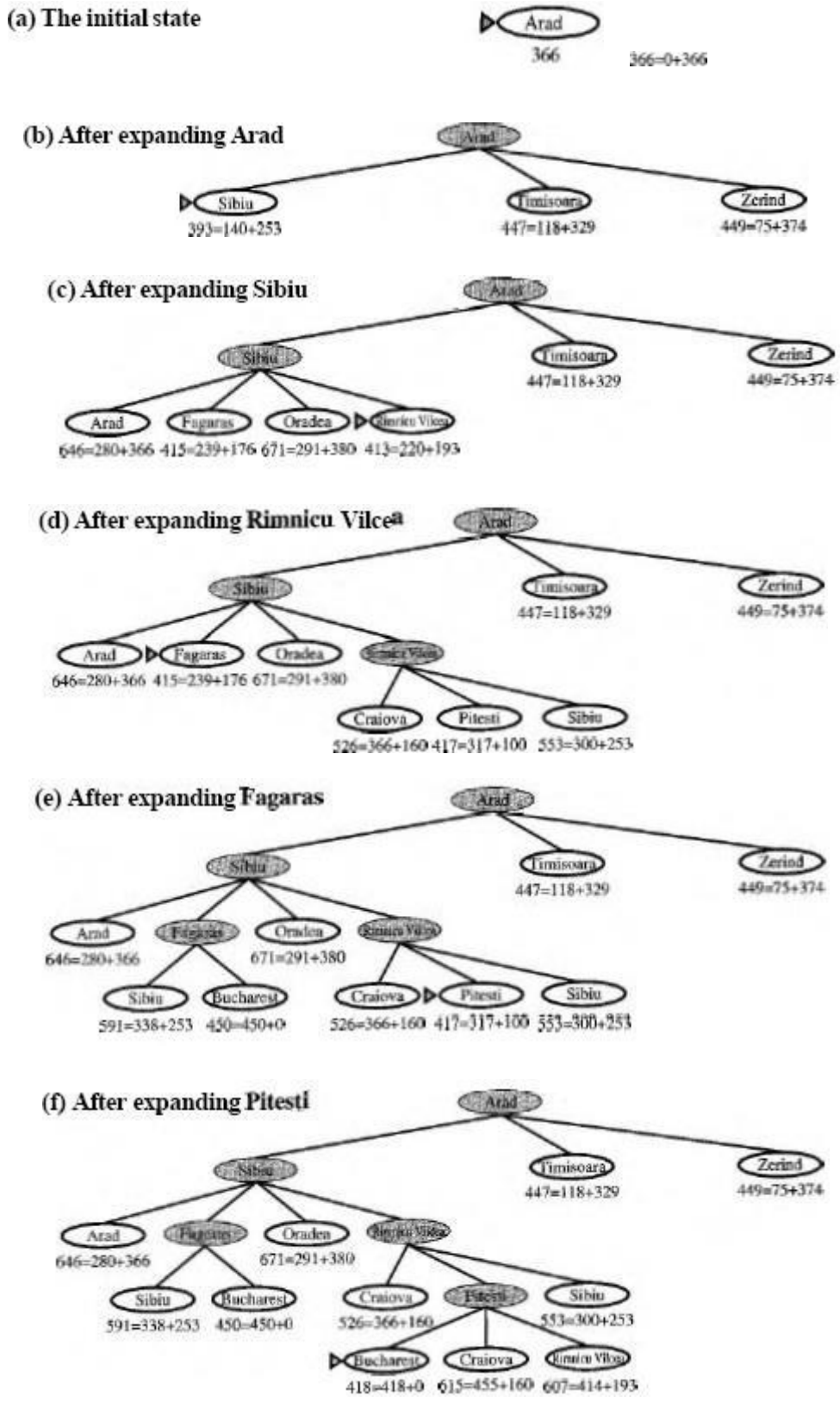
**Heuristic function :** A good heuristic function for route-finding problems isStraight-Line Distance to the goal and it is denoted as hSLD(n).

hSLD(n) = Straight-Line distance between n and the goal locatation

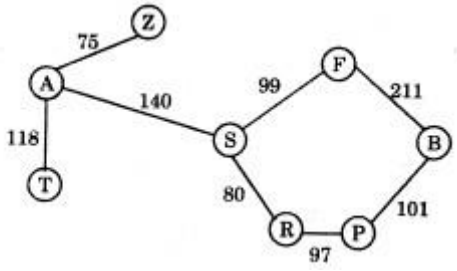
**Values of hSLD-straight-line distances to Bucharest**

| | | | |
|---|---|---|---|
| **Arad** | 366 | **Mehadia** | 241 |
| **Bucharest** | 0 | **Neamt** | 234 |
| **Craiova** | 160 | **Oradea** | 380 |
| **Drobeta** | 242 | **Pitesti** | 100 |
| **Eforie** | 161 | **Rimnicu Vilcea** | 193 |
| **Fagaras** | 176 | **Sibiu** | 253 |
| **Giurgiu** | 77 | **Timisoara** | 329 |
| **Hirsova** | 151 | **Urziceni** | 80 |
| **Iasi** | 226 | **Vaslui** | 199 |
| **Lugoj** | 244 | **Zerind** | 374 |

**Stages in an A* search for Bucharest. Nodes are labeled with f (n) = g(n) + h(n)**




**Example 2 : Finding the path from one node to another node**

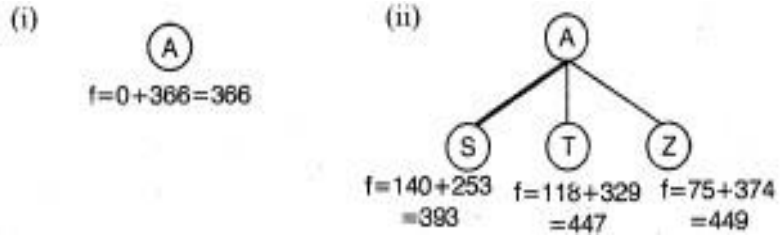



***Straight - line distance to B from A:***

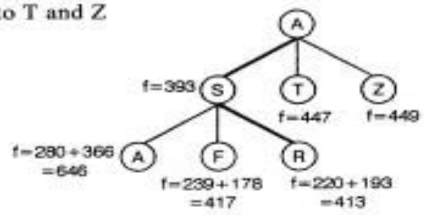
A - 366
B - 0
F - 178
P - 98
R - 193
S - 253
T - 329
Z - 374

**Solution:**

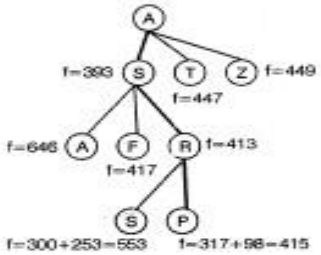
From the given graph and estimated cost, the goal state is identified as B from AApply the evaluation function f(n) = g(n) +h(n) to find a path from A to B



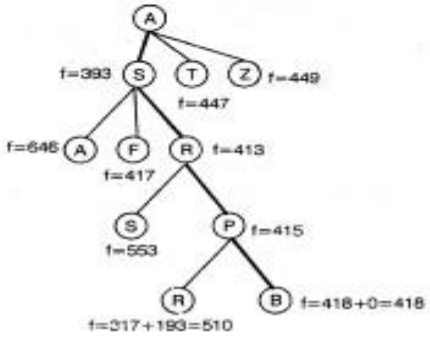
(iii) S is selected for next level of expansion, since f(S) is minimum from S, when comparing to T and Z




(iv) R is selected for next level of expansion, since f(R) is minimum when comparing to A and F.




(v) P is selected for next level of expansion. since f(P) is minimum.




From P, goal state B is reached. Therefore the path from A to B using A* search is A – S - R - P -B : 418 (ie) {140 + 80 + 97 + 101), that the path cost is less than Greedy search path cost.

**Time and space complexity:** Time complexity depends on the heuristic function and the admissible heuristic value. Space complexity remains in the exponential order.

**The behavior of A* search Monotonicity (Consistency**)

In search tree any path from the root, the f- cost never decreases. This conditionis true for almost all admissible heuristics. A heuristic which satisfies this property is called monotonicity(**consistency**).

**A** heuristic h(n) is consistent if, for every node n and every successor n' of n generated by any action a, the estimated cost of reaching the goal from n is no

greater than the step cost of getting to n' plus the estimated cost of reaching thegoal from n':

If the heuristic is non-monotonic, then we have to make a minor correction that restores monotonicity.

**Example for monotonic**

Let us consider two nodes n and n', where n is the parent of n'


**For example**

g(n) = 3 and h(n) = 4. then f(n) = g(n) + h(n) = 7.
g(n') = 54 and h(n') = 3. then f(n') = g(n') + h(n') = 8

**Example for Non-monotonic**

Let us consider two nodes n and n', where n is the parent of n'. For example


g(n) = 3 and h(n) = 4. then f(n) = g(n) + h(n) = 7.
g(n') = 4 and h(n') = 2. then f(n') = g(n') + h(n') = 6.

To reach the node n the cost value is 7, from there to reach the node n' the value of cost has to increase as per monotonic property. But the above example does not satisfy this property. So, it is called as non-monotonic heuristic.

**How to avoid non-monotonic heuristic?**

We have to check each time when we generate anew node, to see if its f-cost is less that its parent's f-cost; if it is we have to use the parent's f- cost instead.

Non-monotonic heuristic can be avoided using path-max equation.

f(n') = max (f{n), g(n') + h(n'))

**Optimality**

A* search is complete, optimal, and optimally efficient among all algorithmsA* using GRAPH-

SEARCH is optimal if h(n) is consistent.

**Completeness**

A* is complete on locally finite graphs (graphs with a finite branching factor)provided there is some constant d such that every operator costs at least d.

**Drawback**

A* usually runs out of space because it keeps all generated nodes in memory

**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" (IDA*) algorithm.

The memory requirements of A* is reduced by combining the heuristic function with iterative deepening resulting an IDA* algorithm.

**Iterative Deepening A* search (IDA*)**

Depth first search is modified to use an f-cost limit rather than a depth limit for IDA* algorithm.

Each iteration in the search expands all the nodes inside the contour for the current f-cost and moves to the next contour with new f - cost.

**Space complexity** is proportional to the longest path of exploration that is bd isa good estimate of storage requirements

**Time complexity** depends on the number of different values that the heuristic function can take on

Optimality: yes, because it implies A* search. Completeness: yes, because it implies A* search.

Disadvantage: It will require more storage space in complex domains (i.e) Each contour will include only one state with the previous contour. To avoid this, we increase the f-cost limit by a fixed amount  on each iteration, so that the total number of iteration is proportional to 1/ . Such an algorithm is called  admissible.

The two recent memory bounded algorithms are:

- Recursive Best First Search (RBfS)
- Memory bounded A* search (MA*)

**Recursive Best First Search (RBFS)**

A recursive algorithm with best first search technique uses only linear space.
It is similar to recursive depth first search with an inclusion (i.e.) keeps track of the f-value of the best alternative path available from any ancestor of the currentnode.

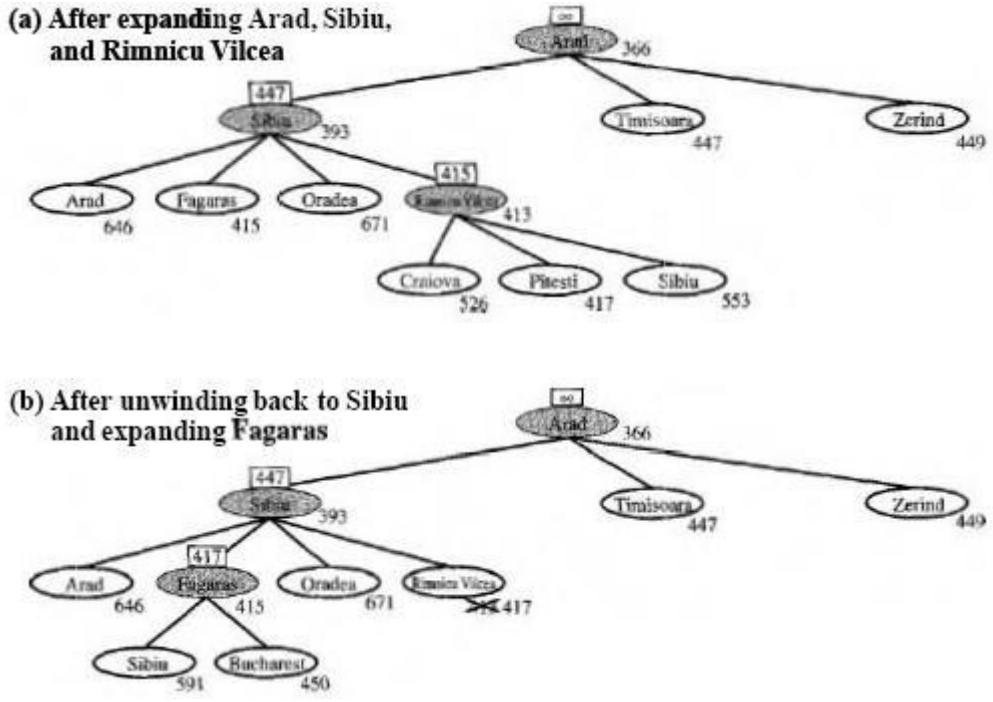
If the current node exceeds this limit, the recursion unwinds back to the alternative path and replaces the f-value of each node along the path with the best f-value of its children.

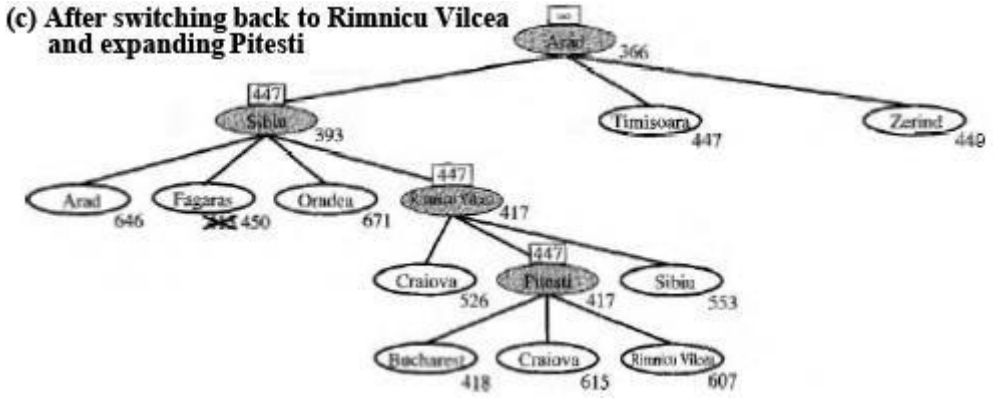
The main idea lies in to keep track of the second best alternate node (forgotten node) and decides whether it's worth to reexpand the subtree at some later time.

## Algortihm For Recursive Best-First Search

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a
solution, or failure
RBFS(problem,MAKE-NODE(INITIAL-STATE[problem]),  ∞ )
function,   RBFS(problem,   node,   f_limit)   returns   a
solution, or failure and a new f-cost limit
if GOAL-TEST[problem](state) 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
```

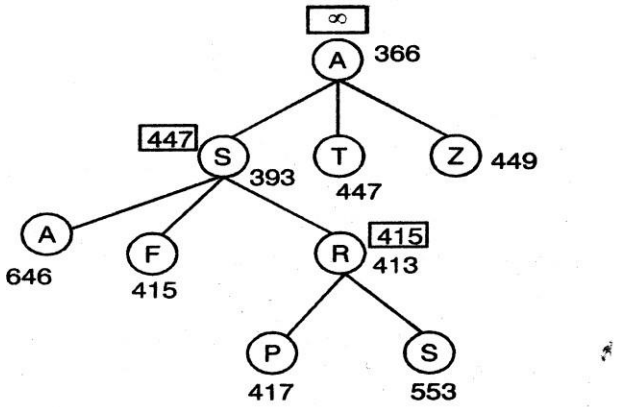
## Stages in an RBFS search for the shortest route to Bucharest.





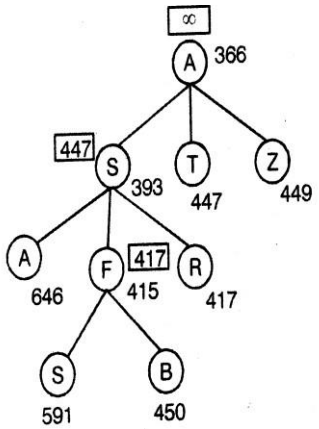

Example:

a) After expanding A, S, and R, the current best leaf(P) has a value that is worsethan the best alternative path (F)



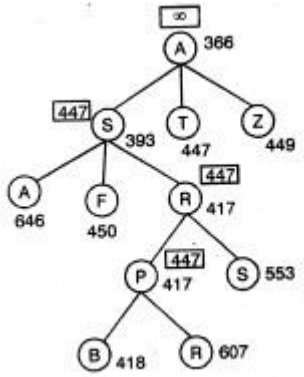

f-limit value of each recursive call is shown on top of each current node. After expanding R, the condition f[best] >f-limit (417 > 415) is true and returns f[best] to that node.

b) After unwinding back to and expanding F




Here the f[best] is 450 and which is greater than the f-limit of 417. Therefore if returns and unwinds with f[best] value to that node.

c) After switching back to Rand expanding P.




The best alternative path through T costs at least 447, therefore the path through R and P is considered as the best one.

**Time and space complexity :** RBFS is an optimal algorithm if the heuristic function h(n) is admissible. Its time complexity depends both on the accuracy of the heuristic function and on how often the best path changes as nodes are expanded. Its space complexity is O(bd), even though more is available.

A search techniques (algorithms) which uses all available memory are:

a. MA* (Memory - bounded A*)
b. SMA* (Simplified MA*)

**2.4.3 Simplified Memory - bounded A* search (SMA*)**

SMA* algorithm can make use of all available memory to carry out the search.

**Properties of SMA* algorithm:**

(a) It will utilize whatever memory is made available to it.
(b) It avoids repeated states as far as its memory allows.

It is **complete** if the available memory is sufficient to store the deepest solutionpath.
It is optimal if enough memory is available to store the deepest solution path. Otherwise, it returns the best solution that can be reached with the available memory.

**Advantage:** SMA* uses only the available memory.

**Disadvantage:** If enough memory is not available it leads to unoptimalsolution.

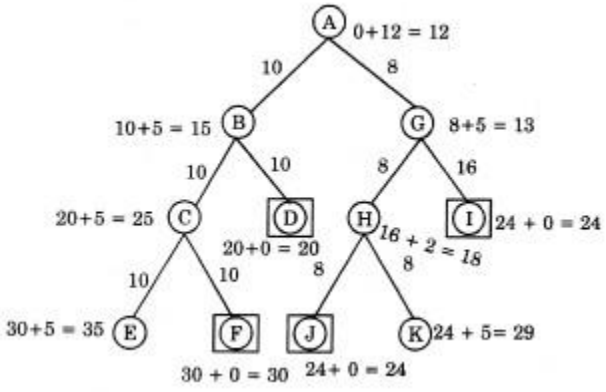
**Space and Time complexity:** depends on the available number of node.

**The SMA* Algorithm**

function SMA*(*problem*) returns a solution sequenceinputs: *problem,* a problem
local variables: *Queue,* a queue of nodes ordered by *f-cost*

Queue<-MAKE-QUEUE(*{MAKE-NODE(INITIAL-STATE[problem])}* )
loop do
if *Queue* is empty then return failuren<-deepest least-f-cost node in *Queue*if *GOAL-TEST(n)* then return success s<-NEXT-*SUCCESSOR(n)*
if *s* is not a goal and is at maximum depth thenf{s)<- 
else
*f{s)*<- *MAX(f(n),* g(s)+h(s))
if all of *n's* successors have been generated then
update *n's* f-cost and those of its ancestors if necessaryif *SUCCESSORS(n)* all in the memory then remove *n* from*Queue*
if memory is full then
delete shallowest, highest-f-cost node in *Queue*remove it from its parent's successor list insert its parent on *Queue* if necessary
insert *s* on *Queue*
end

Example:




The values at the nodes are given as per the A* function i.e. g+h=f

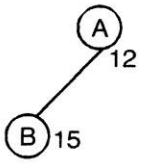
From the Figure we identified that the goal states are D,F,J,I because the h valueof these nodes are zero (marked as a square)

Available memory - 3 nodes of storage space.

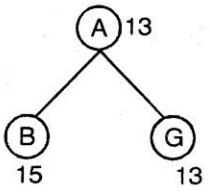
**Task:** Find a optimal path from A to anyone of the goal state.Solution:

(i) 

Initial state

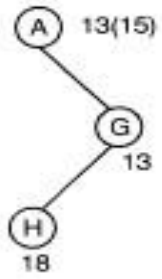
(ii) 

One successor to be added at each stage, therefore B is added to A.

(iii) 

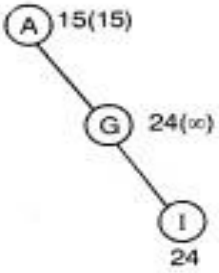
Now, all the children of A is expanded, update the f – cost of root node to the minimum of its children, that is f(A) is 13, the memory is now full.

(iv)



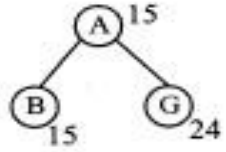

Now, expand the node G to its next successor, but already the memory is full. B is discarded from the queue and it is added as a forgotten descendent of A, which is shown in the parenthesis. The node H is added to G, with $f(H)=18$ but H is not a goal state and it uses all the available memory. Hence there is no way to find a solution through H, make $f(H) = \infty$

(v)



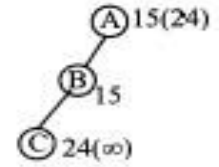

G is expanded again. We drop H and add I (i.e) $f(I) = 24$. The value of G is updated with minimum of H and I and the forgotten value in the parenthesis (i.e) $f(G) = 24$ $(\infty)$ Again the value of A is also updated with minimum of $f(B) = 15$ and $f(G) = 24$ and the forgotten value in the parenthesis (i.e) $f(A) = 15(15)$. Notice that I is a goal node, but it might not be the best solution because A's $f$–cost is only 15, it leads a path in another direction.

(vi)



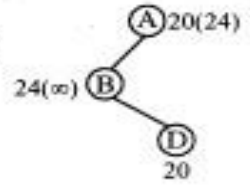

The path from the root (A) leads to B because f(A) = 15 in the previous step. Therefore B is generated for the second time.

(vii)




From B, C is generated, which is a non goal state (i.e) $f(C) = \infty$

(viii)



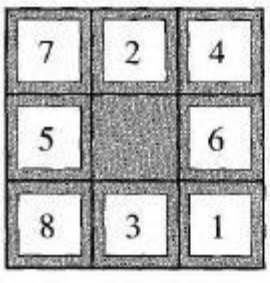

Drop C, add the another successor (D) to B (i.e) $f(D) = 20$, and this value is inherited by B and A.

Now, the deepest, lowest f - cost node is D and it is a goal state also, the search terminates with D as the goal state path=A–B–D.

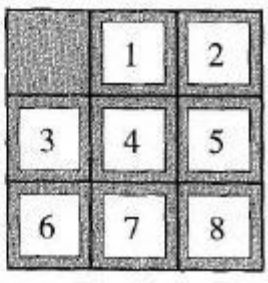
## 2.5 HEURISTIC FUNCTIONS

The 8-puzzle was one of the earliest heuristic search problems.

**Given :**




Start State




Goal State

**Task :** Find the shortest solution using heuristic function that never over estimates the number of steps to the goal.

**Solution :** To perform the given task two candidates are required, which are named as h1 and h2

h1 = the number of misplaced tiles.

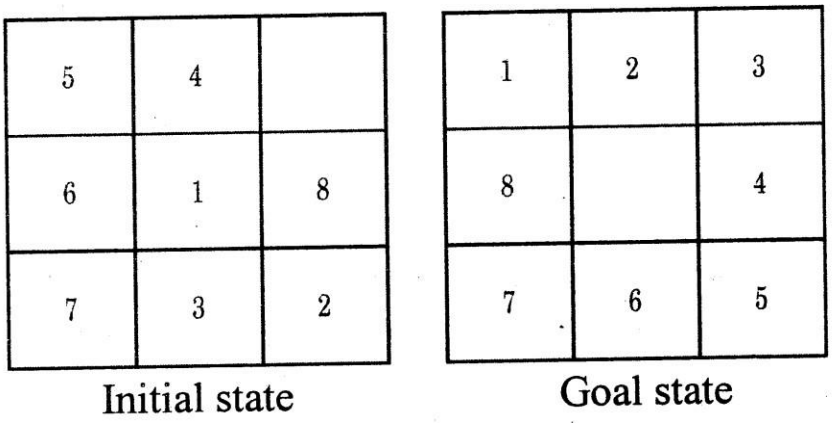
All of the eight tiles are out of position in the above figure, so the start state would have hl = 8. hl is an admissible heuristic, because it is clear that any tile that is out of place must be moved at least once.

h2 = the sum of the distances of the tiles from their goal positions. Because tiles cannot move along diagonals, the distance we will count is the sum of the horizontal and vertical distances. This is called as the **city block distance** or **Manhattan distance.** h2 is also admissible, because any move can do is move one tile one step closer to the goal. Tiles 1 to 8 in the start state give a Manhattan distance of

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

True solution cost is h1 + h2 = 26Example :

h1=7




h2 = 2 + 3 + 3 + 2 + 4 + 2 + 0 + 2 = 18

True Solution Cost is h1 + h2 = 25

**Effective branching factor(b*)**

In the search tree, 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 N nodes. Thus:

$N= 1+ b^* + (b^*)^2 + (b^*)^3 + +(b^*)^d$Example:

For example, if **A*** finds a solution at depth 5 using 52 nodes, then the effectivebranching factor is 1.92.

Depth = 5
N = 52

Effective branching factor is 1.92.

**Relaxed problem**

A problem with less restriction on the operators is called a relaxed problem. If the given problem is a relaxed problem then it is possible to produce good heuristic function. **Example:** 8 puzzle problem, with minimum number of operators.

**2.6 Local Search Algorithms And Optimization Problems**

In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution.

The best state is identified from the objective function or heuristic cost function. In such cases, we can use local search algorithms (ie) keep only a single currentstate, try to improve it instead of the whole search space explored so far

For example, in the 8-queens problem, what matters is the final configuration ofqueens, not the order in which they are added.

**Local search** algorithms operate a single **current state** (rather than multiple paths) and generally move only to neighbors of that state. Typically, the paths followed by the search are not retained.

They have two key advantages:

(1) They use very little memory-usually a constant amount; (2) They can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.

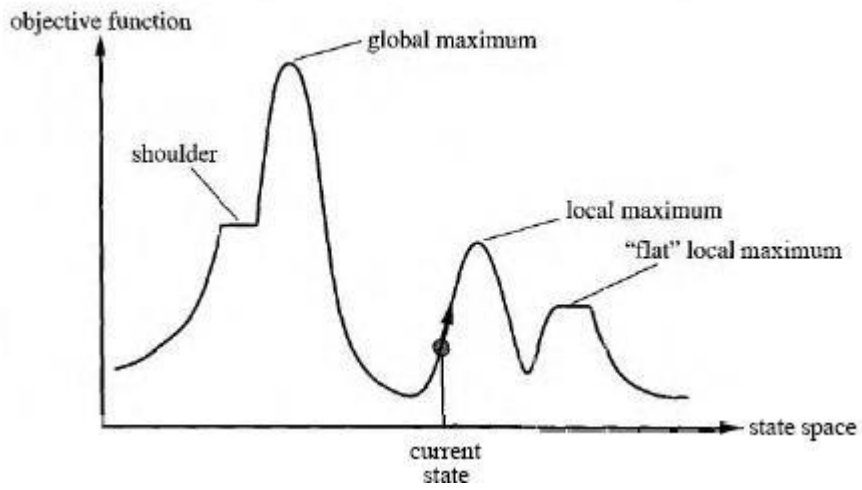
The local search problem is explained with the state space land scape. A landscape has:

**Location** - defined by the state

**Elevation** - defined by the value of the heuristic cost function or objective function, if elevation corresponds to cost then the lowest valley (global minimum) is achieved. If elevation corresponds to an objective function, then thehighest peak (global maximum) is achieved.

A complete local search algorithm always finds a goal if one exists, an optimal algorithm always finds a global minimum/maximum.

**A one-dimensional state space landscape in which elevation corresponds to the objective function.**




**Applications**

Integrated - circuit designFactory - floor layout
Job-shop scheduling
Automatic programming
Vehicle routing
Telecommunications
network Optimization

**Advantages**

- Constant search space. It is suitable for online and offline search

- ❖ The search cost is less when compare to informed search
- ❖ Reasonable solutions are derived in large or continuous state space forwhich systematic algorithms are unsuitable.

**Some of the local search algorithms are:**

1. Hill climbing search (Greedy Local Search)
2. Simulated annealing
3. Local beam search
4. Genetic Algorithm (GA)

## 2.6.1 Hill Climbing Search (Greedy Local Search)

The **hill-climbing** search algorithm is simply a loop that continually moves in the direction of increasing value. It terminates when it reaches a "peak" where no neighbor has a higher value. The algorithm does not maintain a search tree, so the current node data structure need only record the state and its objective function value. At each step the current node is replaced by the best neighbor;

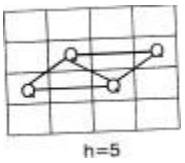
To illustrate hill-climbing, we will use the 8-queens, where each state has 8 queens on the board, one per column. The successor function returns all possible states generated by moving a single queen to another square in the same column (so each state has 8 x 7 = 56 successors).

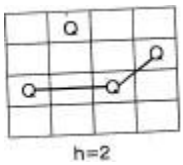
Hill-climbing algorithms typically choose randomly among the set of best successors, if there is more than one.

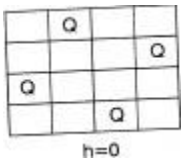
The heuristic cost function h is the number of pairs of queens that are attacking each other, either directly or indirectly.
Hill-**climbing** search algorithm

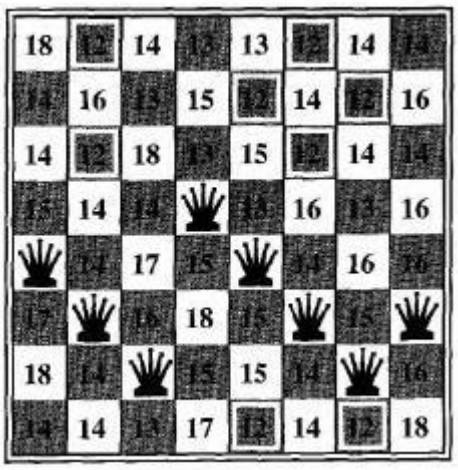
```
function HILL-CLIMBING(problem) returns a state that is a
local maximum
inputs: problem, a problem
local variables:current, a node and neighbor, a node
current <- MAKE-NODE(INITIAL-STATE[problem])
loop do
neighbor <- a highest-valued successor of current
if VALUE[neighbor] <= VALUE[current] then
return STATE[current]
current <- neighbor
```

The global minimum of this function is zero, which occurs only at perfectsolutions.



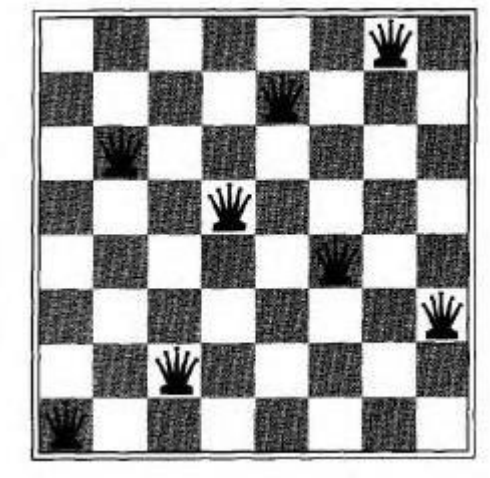



An 8-queens state with heuristic cost estimate h = 17, showing the value of h foreach possible successor obtained by moving a queen within its column. The bestmoves are marked.

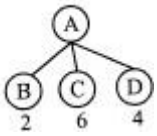
A local minimum in the 8-queens state space; the state has h = 1 but every successor has a higher cost.



Hill climbing often gets stuck for the following reasons:

**Local maxima or foot hills :** a local maximum is a peak that is higher than each of its neighboring states, but lower than the global maximum

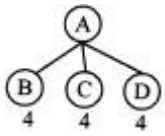
Example :




The evaluation function value is maximum at C and from their there is no path exist for expansion. Therefore C is called as local maxima. To avoid this state, random node is selected using back tracking to the previous node.

**Plateau or shoulder:** a plateau is an area of the state space landscape where the evaluation function is flat. It can be a flat local maximum.

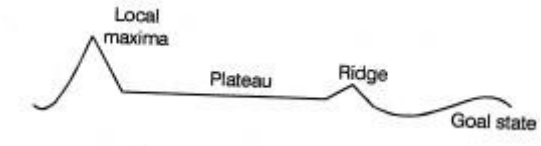
Example :




The evaluation function value of B C D are same, this is a state space of plateau. To avoid this state, random node is selected or skip the level (i.e) select the node in the next level

Ridges: Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate. But the disadvantage is more calculations to be done function

**Structure of hill climbing drawbacks**




**Variants of hill-climbing**

**Stochastic hill climbing -** Stochastic hill climbing chooses at random from among the uphill moves; the probability of selection can vary with the steepnessof the uphill move.

**First-choice hill climbing -** First-choice hill climbing implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state

**Random-restart hill climbing -** Random-restart hill climbing adopts the well known adage, "If at first you don't succeed, try, try again." It conducts a series of hill-climbing searches from randomly generated initial state, stopping when a goal is found.

**2.6.2 Simulated annealing search**

An algorithm which combines hill climbing with random walk to yield both efficiency and completeness

In metallurgy, **annealing** is the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them

When the search stops at the local maxima, we will allow the search to take some down Hill steps to escape the local maxima by allowing some "bad" movesbut gradually decrease their size and frequency. The node is selected randomly and it checks whether it is a best move or not. If the move improves the situation, it is executed.  E variable is introduced to calculate the probability of worsened. A Second parameter T is introduced to determine the probability.

**The simulated annealing search algorithm**

```
function SIMULATED-ANNEALING(problem, schedule) returns a
solution state
inputs: problem, a problem
schedule, a mapping from time to "temperature"
local variables: current, a node
next, a node
T, a "variable" 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
ΔE <- VALUE[next] - VALUE[current]
if  ΔE > 0  then current <- next
else current <- next only with probability e^(ΔE/T)
```

**Property of simulated annealing search**

T decreases slowly enough then simulated annealing search will find a globaloptimum with probability approaching one

**Applications**

VLSI layout Airline scheduling

**2.6.3 Local beam search**

Local beam search is a variation of beam search which is a path based algorithm.It uses K states and generates successors for K states in parallel instead of one state and its successors in sequence. The useful information is passed among theK parallel threads.

The sequence of steps to perform local beam search is given below:

- Keep track of K states rather than just one.
- Start with K randomly generated states.
- At each iteration, all the successors of all K states are generated.
- If anyone is a goal state stop; else select the K best successors from thecomplete list and repeat.

This search will suffer from lack of diversity among K states.

Therefore a variant named as stochastic beam search selects K successors at random, with the probability of choosing a given successor being an increasing function of its value.

## 2.6.4 Genetic Algorithms (GA)

**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

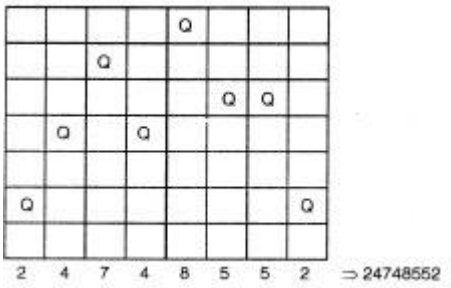
GA begins with a set of k randomly generated states, called the **population.**
Each state, or **individual,** is represented as a string over a finite alphabet.

For Example an 8 queen's state could be represented as 8 digits, each in the range from 1 to 8.

**Initial population:** K randomly generated states of 8 queen problem

**Individual (or) state:** Each string in the initial population is individual (or) state. In one state, the position of the queen of each column is represented.

**Example:** The state with the value 24748552 is derived as follows:




The Initial Population (Four randomly selected States) are :

24748552

32752411

24415124

32543213

**Evaluation function (or) Fitness function:** A function that returns higher values for better State. For 8 queens problem the number of non attacking pairs of queens is defined as fitness function

Minimum fitness value is 0

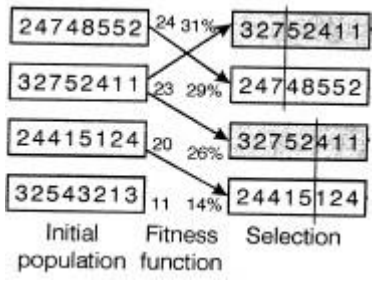
Maximum fitness value is : 8*7/2 = 28 for a solutionThe values of the four states are 24, 23, 20,

and 11.

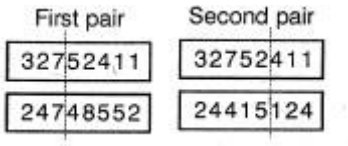
The probability of being chosen for reproduction is directly proportional to thefitness score, which is denoted as percentage.

24 / (24+23+20+11) = 31%
23 / (24+23+20+11) = 29%
20 / (24+23+20+11) = 26%
11 / (24+23+20+11) = 14%

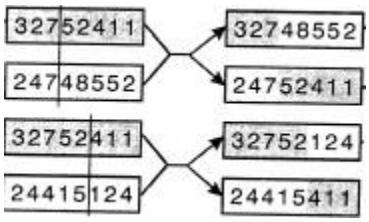
**Selection** : A random choice of two pairs is selected for reproduction, by considering the probability of fitness function of each state. In the example one state is chosen twice probability of 29%) and the another one state is not chosen (Probability of 14%)



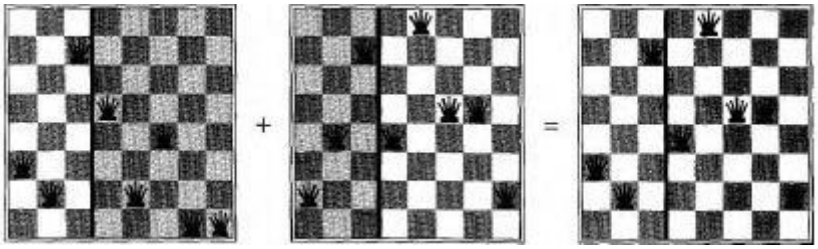

**Cross over:** Each pair to be mated, a crossover point is randomly chosen. For the first pair the crossover point is chosen after 3 digits and after 5 digits for thesecond pair.



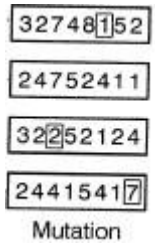

**offspring :** Offspring is created by crossing over the parent strings in the crossover point. That is, the first child of the first pair gets the first 3 digits from the first parent and the remaining digits from the second parent. Similarly the second child of the first pair gets the first 3 digits from the second parent and the remaining digits from the first parent.




The 8-queens states corresponding to the first two parents




**Mutation :** Each location is subject to random **mutation** with a small independent probability. One digit was mutated in the first, third, and fourth offspring



- Evaluation function (fitness function) is applied to find better stateswith higher values.
- Produce the next generation of states by selection, crossover andmutation

## 2.8 CONSTRAINT SATISFACTION PROBLEMS(CSP)

Constraint satisfaction problems (CSP) are mathematical problems where one must find states or objects that satisfy a number of constraints or criteria. A constraint is a restriction of the feasible solutions in an optimization problem.

Some examples for CSP's are:The n-queens problem
A crossword puzzle
A map coloring problem
The Boolean satisfiability problem

A cryptarithmetic problem

All these examples and other real life problems like time table scheduling, transport scheduling, floor planning etc. are instances of the same pattern,

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.

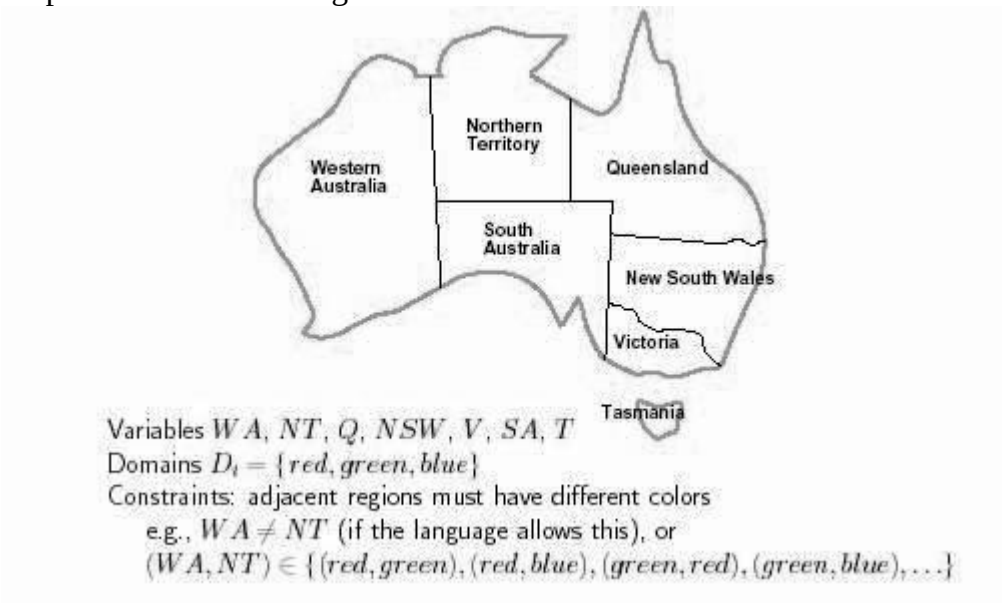
A **State** of the problem is defined by an **assignment** of values to some or all ofthe variables,{Xi = vi, Xj = vj,…}. An assignment that does not violate any constraints is called a **consistent** or **legal assignment.**

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 :**

The map coloring problem. The task of coloring each region red, green or blue insuch a way that no neighboring regions have the same color.

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 away that the neighboring regions must not 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

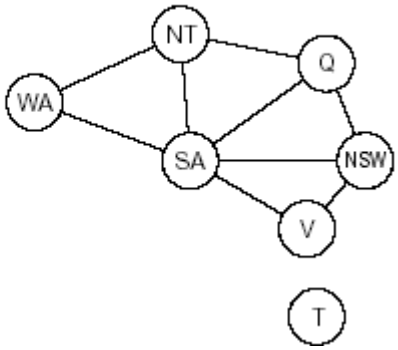
represented as the inequality WA  NT) There are many possible solutions, such as

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

Constraint Graph : A CSP is usually represented as an undirected graph, called constraint graph where the nodes are the variables and the edges are the binaryconstraints.

The map-coloring problem represented as a constraint graph.

Constraint graph: nodes are variables, arcs show constraints




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 nif there are n variables. So Depth first search algorithms are popular for CSPs.

## Varieties of CSPs

### Discrete variables

Discrete variables can have

- Finite Domains
- Infinite domains

### 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 $O(d^n)$ – that is, exponential in the number of variables.

Finite domain CSPs include **Boolean CSPs**, whose variables can be either true or false.

### Infinite domains

Discrete variables can also have **infinite domains** – for example, the set of integers or the set of strings. With infinite domains, it is no longer possible to describe constraints by enumerating all allowed combination of values. For example, if Jobl, which takes five days, must precede Jobs, then we would need a constraint language of algebraic inequalities such as

Startjob1 + 5 <= Startjob3.

### 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 :**

**Unary constraints –** Which restricts a single variable.Example : SA  green

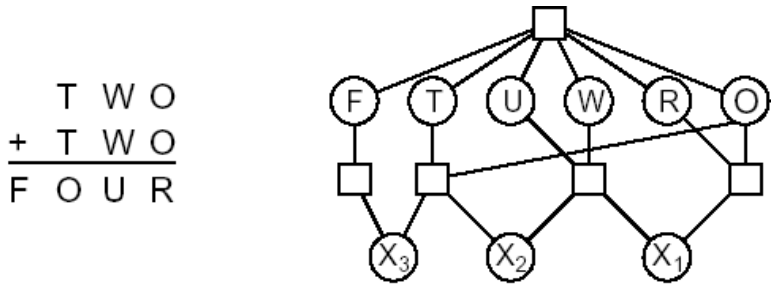
**Binary constraints** - relates pairs of variables.Example : SA  WA

**Higher order constraints** involve 3 or more variables.

Example : cryptarithmetic puzzles. Each letter stands for a distinct digit

The aim is to find a substitution of digits for letters such that the resulting sum isarithmetically correct, with the added restriction that no leading zeros areallowed.

**Constraint graph for the cryptarithmetic Problem**




Variables: $F\ T\ U\ W\ R\ O\ X_1\ X_2\ X_3$
Domains: $\{0,1,2,3,4,5,6,7,8,9\}$
Constraints
$alldiff(F,T,U,W,R,O)$
$O + O = R + 10 \cdot X_1$, etc.

Alldiff constraint can be broken down into binary constraints - F  T, F  U, andso on.
The addition constraints on the four columns of the puzzle also involve severalvariables and can be written as

O + O = R + 10 . X1
X1 + W + W = U + 10 . X2X1 + T + T = O + 10 . X3 X3 = F

Where X1, X2, and X3 are **auxiliary variables** representing the digit (0 or 1)carried over into the next column.

Real World CSP's : Real world problems involve read-valued variables,

- Assignment problems Example : who teaches what class.
- Timetabling Problems Example : Which class is offered when & where?
- Transportation Scheduling
- Factory Scheduling

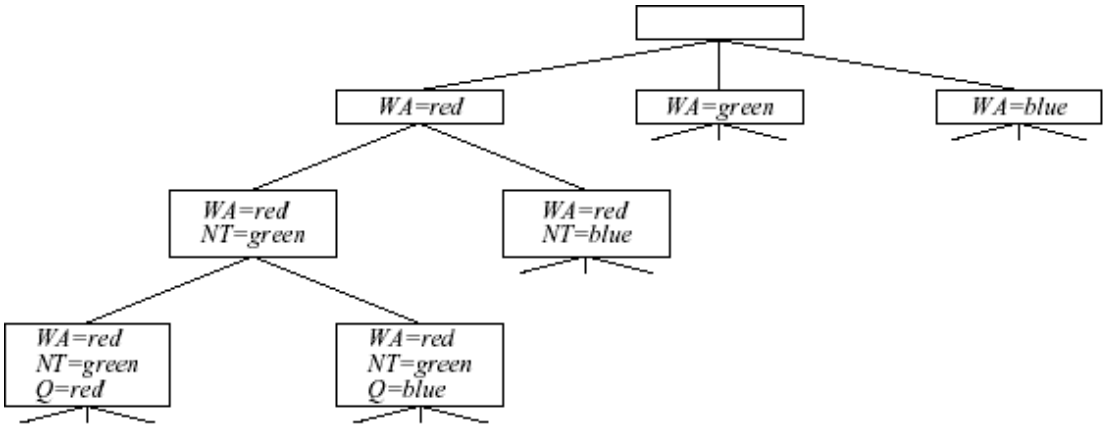
**Backtracking Search for CSPs**

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.

```
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
```

**Part of search tree generated by simple backtracking for the map coloring problem**




Improving backtracking efficiency is done with general purpose methods, which can give huge gains in speed.

- Which variable should be assigned next and what order should be tried?
- What are the implications of the current variable assignments for the otherunassigned variables?
- Can we detect inevitable failure early?

**Variable & *value ordering:*** In the backtracking algorithm each unassigned variable is chosen from minimum Remaining Values (MRV) heuristic, that is choosing the variable with the fewest legal values. It also has been called the "most constrained variable" or "fail-first" heuristic.

If the tie occurs among most constrained variables then most constraining variable is chosen (i.e.) choose the variable with the most constraints on remaining variable. Once a variable has been selected, choose the least constraining value that is the one that rules out the fewest values in the remaining variables.

## 2.9 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 forX.

**The progress of a map-coloring search with forward checking.**

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

In forward checking WA = red is assigned first; then forward checking deletes red from the domains of the neighboring variables NT and SA. After Q = green, green is deleted from the domains of NT, SA, and NSW. After V = blue, blue isdeleted from the domains of NSW and SA, leaving SA with no legal values. NT and SA cannot be blue

**Constraint propagation**

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

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

**Constraint** propagation repeatedly enforces constraints locally to detect inconsistencies. This propagation can be done with different types of consistencytechniques. They are:

Node consistency (one consistency)Arc consistency (two consistency) Path consistency (K-consistency)

**Node consistency**

- Simplest consistency technique
- The node representing a variable V in constraint graph is node consistent if for every value X in the current domain of V, each unary constraint on V is satisfied.
- The node inconsistency can be eliminated by simply removing those values from the domain D of each variable V that do not satisfy unary constraint on V.

**Arc Consistency**

The idea of **arc consistency** 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 thecurrent 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.

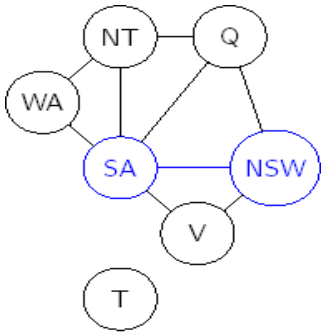



Figure: Australian Territories

In the constraint graph, binary constraint corresponds to arc. Therefore this typeof consistency is called arc consistency.

Arc (vi, vj) is arc consistent if for every value X the current domain of vi there is some value Y in the domain of vj such vi =X and vj=Y is permitted by the binary constraint between vi and vj

Arc-consistency is directional ie if an arc (vi, vj) is consistent than it does not automatically mean that (vj, vi) is also consistent.

An arc (vi, vj) can be made consistent by simply deleting those values from the domain of Di for which there is no corresponding value in the domain of Dj suchthat the binary constraint between Vi and vj is satisfied - It is an earlier detection of inconstency that is not detected by forward checking method.

The different versions of Arc consistency algorithms are exist such as AC-I, AC2,AC-3, AC-4, AC-S; AC-6 & AC-7, but frequently used are AC-3 or AC-4.

**AC - 3 Algorithm**

In this algorithm, queue is used to cheek the inconsistent arcs.When the queue is not empty do the

following steps:

- Remove the first arc from the queue and check for consistency.

- If it is inconsistent remove the variable from the domain and add a newarc to the queue
- Repeat the same process until queue is empty

```
function AC-3( csp) returns the CSP, possibly with
reduced domains
inputs: csp, a binary CSP with variables {X1, X2, . . . ,
Xn}
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-VALUEXS(xi,xj) then
for each Xk in NEIGHBORS[Xj) do
add (Xk , Xi)to queue
```

```
function REMOVE-INCONSISTENT-VALUEXS(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]removed <- true
return removed
```

**k-Consistency (path Consistency)**

A CSP is k-consistent if, for any set of k - 1 variables and for any consistent assignment to those variables, a consistent value can always be assigned to any kth variable

- 1-consistency means that each individual variable by itself is consistent; this is also called node consistency.
- 2-consistency is the same as arc consistency.
- 3-consistency means that any pair of adjacent variables can always be extended to a third neighboring variable; this is also called **path consistency**.

**Handling special constraints**

**Alldiff constraint** - All the variables involved must have distinct values.Example: Crypt

arithmetic problem

The inconsistency arises in Alldiff constraint when m>n (i.e.) m variables are involved in the constraint and n possible distinct values are there. It can be avoided by selecting the variable in the constraint that has a singleton. domain and delete the variable's value from the domain of remaining variables, until the singleton variables are-exist. This simple algorithm will resolve the inconsistency of the problem.

**Resource constraint (Atmost Constraint)** - Higher order constraint or atmost constraint, in which consistency is achieved by deleting the maximum value of any domain if it is not consistent with minimum values; of the other domains.

**2.10 Backtracking Search**

***Chronological backtracking*** **:** When a branch of the search fails, back up to the preceding variable and try a different value for it (i.e.) the most recent decision point is revisited. This will lead to inconsistency in real world problems (map coloring problem) that can't be resolved. To overcome the disadvantage ofchronological backtracking, an intelligence backtracking method is proposed.

***Conflict directed backtracking:*** When a branch of the search fails, backtrack to one of the set of variables that caused the failure-conflict set. The conflict set for variable X is the set of previously assigned variables that are connected to X by constraints. A backtracking algorithm that was conflict sets defined in this wayis called conflict directed backtracking

**Local Search for CSPs**

- Local search method is effective in solving CSP's, because complete stateformulation is defined.
- Initial state - assigns a value to every variable.
- Successor function - works by changing the value of each variable

**Advantage :** useful for online searching when the problem changes.Ex : Scheduling problems

**The *MIN-CONFLICTS* algorithm for solving CSPs by local search**.

```
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
```

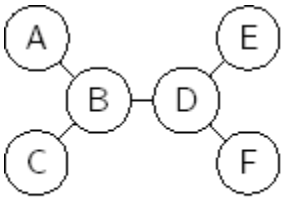
A two-step solution for an &-queens problem using min-conflicts. At each stage, a queen is chosen for reassignment in its column. The number of conflicts (in this case, the number of attacking queens) is shown in each square.


**The structure of problems**

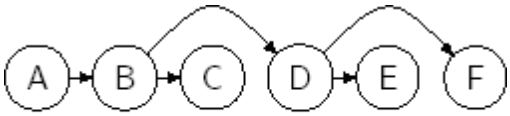
The complexity of solving a CSP-is strongly related to the structure of its constraint graph. If CSP can be divided into independent sub problems, then each sub problem is solved independently then the solutions are combined. When the n variables are divided as n/c subproblems, each will take $d^c$ work to solve. Hence the total work is $O(d^c n/c)$. If n=lO, c=2 then 5 problems are reduced and solved in less time.

Completely independent sub problems are rare, in most cases sub problems of a CSP are connected

The way how to convert the constraint graph of a tree structure CSP into linear ordering of the variables consistent with the tree is shown in Figure. Any two variables are connected by atmost one path in the tree structure




Tree-Structured CSP




Linear ordering

If the constraint graph of a CSP forms a tree structure then it can be solved in linear time number of variables). 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 preceeds it in the ordering label the variables Xl .... Xn in order, every variable except the root has exactly one parent variable.

2. For j from n down 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 Keypoints of this algorithm are as follows:

Step-(2), CSP is arc consistent so the assignment of values in step (3)requires no backtracking.

Step-(2), the arc consistency is applied in reverse order to ensure theconsistency of arcs that are

processed already.

General constraint graphs can be reduced to trees on two ways. They are:

(a) Removing nodes - Cutset conditioning
(b) Collapsing nodes together - Tree decomposition.

**(a) Removing nodes - Cutset conditioning**

- Assign values to some variables so that the remaining variables form atree.

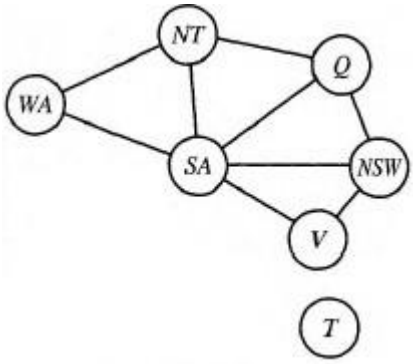
- Delete the value assigned variable from the list and from the domains of the other variables any values that are inconsistent with the value chosenfor the variable.

- This works for binary CSP's and not suitable for higher order constraints.

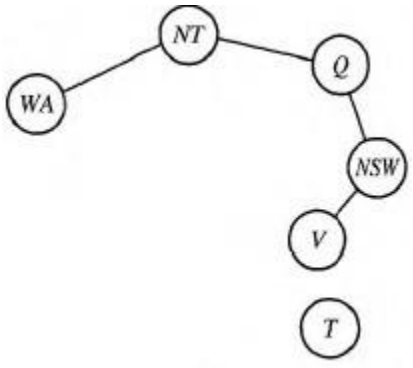
- The remaining problem (tree structure) is solved in linear order timevariables.

Example: In the constraint graph of map coloring problem, the region SA is assigned with a value and it is removed to make the problem in the form of tree structure, then it is solvable in linear time

**The original constraint graph**




**The constraint graph after the removal of SA**




- If the value chosen for the variable to be deleted for tree structure iswrong, then the following algorithm is executed.

***(i)*** Choose a subset S from VARIABLES[CSP] such that the constraintgraph becomes **a tree after removal of *S-cycle cutset.***

(ii) For each variable on S with assignment satisfies all constraints on S.

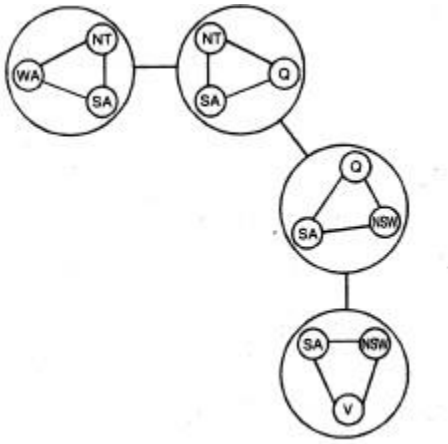
* Remove from the deomains of the remaining variables any values thatare inconsistent with the assignment for S.

* If the remaining CSP has a solution, return it with the assignment for S.

**(b) Collapsing nodes together-Tree decomposition**

- Construction of tree decomposition the constraint graph is divided into a set of subproblems, solved independently and the resulting solutions are combined.
- Works well, when the subproblem is small.
- Requirements of this method are:
    - Every variable in the base problem should appear in atleast one of the subproblem.
    - If the binary constraint is exist, then the same constraint must appear in atleast one of the subproblem. .
    - If the variable appears in two subproblems in the tree, it must appear in every subproblem along the path connecting those subproblems, that is the variable should be assigned with same value and constraint in every subproblem.

**A tree decompositon of the constraint graph**




***Solution***

- If any subproblem has no solution then the entire problem has nosolution.
- If all the sub problems are solvable then a global solution is achieved.

**Adversarial Search**

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

In our terminology, games means deterministic, fully observable environments inwhich there are two agents whose actions must alternate and in which the utilityvalues at the end of the game are always equal and opposite. For example, if one player wins a game of chess (+1), the other player necessarily loses (-1).

There are two types of games

1. **Perfect Information ( Example : chess, checkers)**
2. **Imperfect Information ( Example : Bridge, Backgammon)**

In game playing to select the next state, search technique is required. Game playing itself is considered as a type of search problems. But, how to reduce thesearch time to make on a move from one state to another state.

The **pruning technique** allows us to ignore positions of the search tree that make no difference to the final choice.

**Heuristic evaluation function** allows us to find the utility (win, loss, and draw) of a state without doing a complete search.