Problem Solving using Search.

Search through a state space involves - a set of states, operators and their costs, start state and a test to check for goal state.

The basic search algorithm.

We need to denote the states that have been generated. We call these as nodes. The data structure for a node will keep track of not only the state but also the parent state or the operator that was applied to get this state.

Fringe is a list of nodes that have been generated but not yet explored. It represents the frontier of the search tree generated. Initially, it contains a single node corresponding to the ~~state~~ state.

The algorithm always picks the first node from the fringe for expansion. If the node contains a goal ~~node~~ state, the path to the goal is returned. The path corresponding to a goal node can be found by following the parent pointers Otherwise, all the successor nodes are generated and they are added to the fringe. The successors of the current expanded node are put in fringe. The order in which the successors are put in fringe will determine the property of the search algorithm.

Let L be a list containing the initial state (L = fringe).

Loop if L is empty return failure.
  Node ← select (L)
  if Node is a goal
    then return Node (the path from initial state to Node).
    else generate all successors of Node
      and merge the newly generated state into L.

End Loop.

Factors to evaluate problem solving performance of various search strategies.

1. Completeness: The strategy guarantees to find a solution if one exists.
2. Optimality: The solution has the minimal cost.
3. Complexity: (i) Time complexity - time taken (number of nodes expanded, in worst case or average case, to find a solution.
   (ii) Space complexity - space used by the algorithm measured in terms of maximum size of the fringe.

Different search strategies.

- Blind search strategies or uninformed search.
  - depth first search
  - breadth " "
  - iterative deepening search
  - " broadening "
- Informed search.
- Constraint Statisfaction search.
- Adversary search.

In blind search or uninformed search we do not use any extra information about the problem domain. The two common methods are BFS and DFS.

Search Tree - list of all possible paths, eliminating cycles from the paths is the complete search tree from a state space graph. Thus, it is a data structure containing a root node from where the search starts; every node may have zero or more children; if a node X is a child of node Y, node Y is said to be a parent of node X.

Root Node: The node from which the search starts.
Leaf Node: A node in the search tree having no children.
Ancestor / Descendant: X is an ancestor of Y if either X is Y's parent or X is an ancestor of Y's parent. If X is an ancestor of Y then Y is said to be a descendant of X.
Branching Factor: the maximum no. of children of a non-leaf node in a search tree.
Path: A path in a search tree is a complete path if it begins with the start node and ends with a goal node, otherwise it is a partial path.

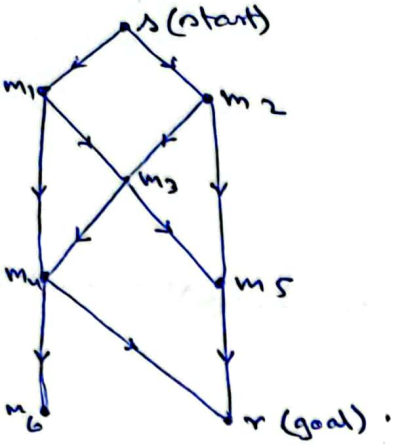
Node Data Structure: A node used in the search algo is a d.s. containing the following:

1. A state description
2. A ptr to the parent of the node.
3. Depth of the node.
4. The operater that generated this node.
5. Cost of this path (sum of operator costs) from the start state.

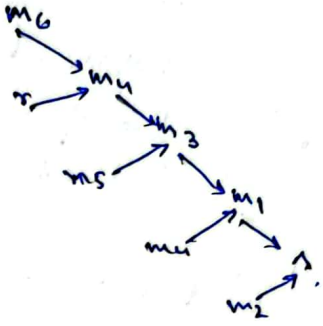
Depth-First Search Method:

```
begin
    put the start node s in OPEN,
    found = false;
    while OPEN not empty and not(found) do
    begin
        remove the topmost node n from OPEN and
            put it in CLOSED;
        if depth of n is equal to depth-bound
            then cleanup CLOSED;
        begin
            expand n, generating all its successors;
            put these successors (in no particular
            order) on top of OPEN except the
            successors already appearing in CLOSED;
            direct backward pointer to n for each
            successor ni of n;
            if any of these successor is goal
            then found = true
            else if any of these successor is a deadend
                then remove it from OPEN and
                    cleanup CLOSED.
        end;
    end;
    if OPEN is empty
    then output "solution path not found"
    else output the solution path obtained by
        tracking back through pointers;
end.
```

Example of DFS.



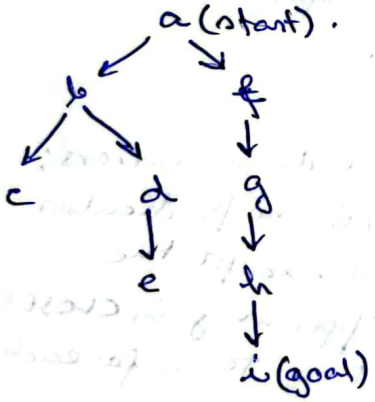
OPEN	CLOSED.
s	
~~s~~	s
m1, m2	s
~~m1~~, m2	s, m1.
m3, m4, m2	s, m1.
~~m3~~, m4, m2	s, m1, m3.
m4, m5, m4, m2	s, m1, m3.
~~m4~~, m5, m4, m2	s, m1, m3, m4
m6, r, m5, m4, m2	s, m1, m2, m4



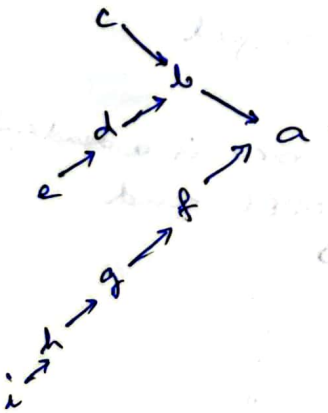
The solution path is $s \rightarrow m_1 \rightarrow m_3 \rightarrow m_4 \rightarrow r$.

" " " depends on the order in which the children are inserted in the OPEN list.

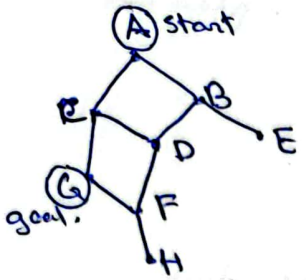
Example:

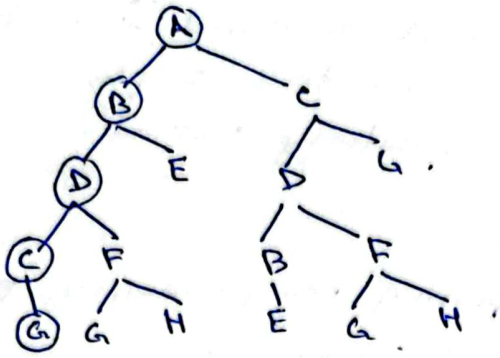


OPEN	CLOSED.
a	
~~a~~	a.
b, f	a
~~b~~, f	a, b
c, d, f	a, b.
~~c~~, d, f	a, b. (cleanup)
d, f	a, b.
~~d~~, f	a, b, d
e, f	a, b, d.
~~e~~, f.	a, b, d, e. (cleanup)
f	a
~~f~~	a, f
g	a, f
~~g~~	a, f, g.
h.	a, f, g.
~~h~~	a, f, g, h.
i	a, f, g, h.



Example:





OPEN	CLOSED
A	
~~A~~	~~A~~ A.
BC	A
~~B~~C	AB.
DEC.	AB.
~~D~~EC	ABD.
CFEC	ABD.
~~G~~FEC	ABDC.
GFEC	ABDC

The solution path is A → B → D → C → G.

Note:

1. Cleanup CLOSED purges from CLOSED all those ancestors of the node that do not have son in OPEN.
   The effects are
   (i) CLOSED will contain only the expanded nodes ~~of~~ on the current path.
   (ii) reduces the size of CLOSE, saves memory.
2. The list CLOSED forms a single path from the start node to the currently expanded node. This restricts the maximum storage which is (depth found * branding factor) in case of a graph OPEN holds the nodes generated in the explicit graph but not yet expanded.
3. For a search graph (not a tree) the depth of a node is the depth of its shallowest parent + 1. Maintaining depth-first in true sense then leads to complication. In practise, DFS takes depth of a node as the depth of its current parent.
4. Node duplications are not strictly checked in practise, only checked against the current path to avoid looping.
5. Effective recursive formulations are possible with lesser amount of storage requirement. For example.

```
procedure dfs(n:node)
begin
    if n is a goal node then terminate = true else
    while not(terminate) and for every successor
        ni of n do dfs(ni).
end;
```

6. Advantages are.

(i) low memory and overhead, easy to implement

(ii) quick solution in certain graphs.

(iii) human problem solving method.

7. Disadvantages are

(i). sometimes exploring non-promising path deeper.

(ii) unnecessary node expansions.

(iii) solution paths not optimal in terms of depth.

(iv). a node may be expanded more than once along various paths in fact it may be exponential in certain cases (as shown in figure).

8. The algorithm takes exponential time. If N is the maximum depth of a node in the search space and d is depth where solution can be found, in the worst case the algorithm will take time $O(b^d)$.

9. The space taken is linear in the depth of the search tree, $O(bN)$, where N is the length of the longest path or maximum depth.

10. If the search tree has infinite depth, the algorithm may not terminate. This can happen if the search space is infinite. It can also happen if the search space contains cycles. The latter case can be handled by checking for cycles in the algorithm. Thus DFS is not complete.

Prolog outline to dfs: Any problem to be solved by dfs can be written as

```
solve(S, Moves) :- solve-dfs(S, [S], Moves).
solve-dfs(S, H, []) :- final-state(S).
solve-dfs(S, H, [M|More]) :- move(S, M), update(S, M, S1),
                             legal(S1), not(member(S1, H)),
                             solve-dfs(S1, [S1|H], More).
```

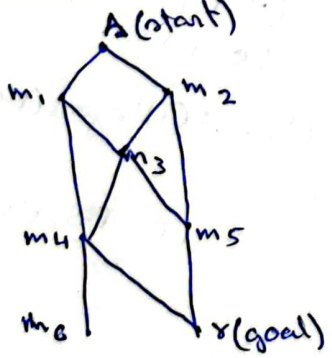
Example: The wolf-goat-cabbage problem.

```
wgc(left, [w, g, c], []).        /* initially */
final_state(wgc(right, [], [w, g, c])).
move(wgc(left, L, R), M) :- member(M, L).
move(wgc(right, L, R), M) :- member(M, R).
move(wgc(B, L, R), alone).
update(wgc(B, L, R), M, wgc(B1, L1, R1)) :-
                update_boat(B, B1),
                updatebanks(M, B, L, R, L1, R1).
update_boat(left, right).
update_boat(right, left).
updatebanks(alone, B, L, R, L, R).
updatebanks(M, left, L, R, L1, R1) :- select(M, L, L1),
                insert(M, R, R1).
updatebanks(M, right, L, R, L1, R1) :- select(M, R, R1),
                insert(M, L, L1).
insert(X, [], [X]).
insert(X, [Y|Ys], [X, Y|Ys]) :- precedes(X, Y).
insert(X, [Y|Ys], [Y|Z]) :- precedes(Y, X),
                insert(X, Ys, Z).
precedes(w, X).
precedes(X, c).
legal(wgc(left, L, R)) :- not(illegal(R)).
legal(wgc(right, L, R)) :- not(illegal(L)).
illegal(F) :- member(w, F), member(g, F).
illegal(F) :- member(g, F), member(c, F).
select(M, F, F1) :- remove(M, F, F1), not(illegal(F1)).
remove(M, [M|Fs], Fs).
remove(M, [X|Fs], [X|FY]) :- remove(M, Fs, FY).
```

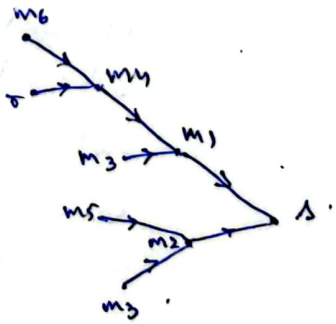
Breadth-first search method (Level by level expansion).

```
begin
    put startnode s in open;
    found = false;
    while OPEN is not empty and not(found) do
    begin
        remove the topmost node n from OPEN and
            put it in CLOSED;
        expand n generating all its successors;
        put these successors (in no particular order)
            at the end of OPEN except the successors
            already appearing in CLOSED.
            /* also in OPEN for graphs */
        direct backwards pointers to n from each
            successor ni;
        if any of these successors is a goal node
        then found = true
        else if one of these successors is a dead end
            then remove it from OPEN;
    end;
    if OPEN is empty
    then output failure message
    else output solution by tracking back through
        pointers;
end.
```

Example:

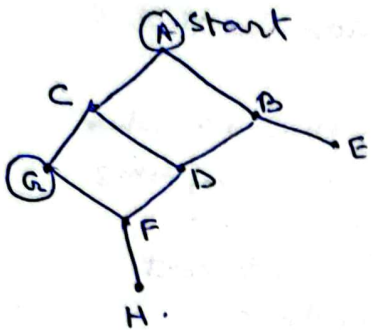


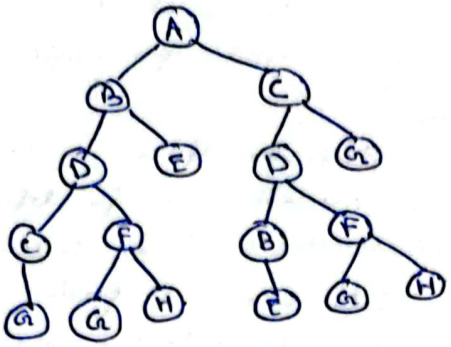
OPEN	CLOSED
s	
~~s~~	s
$m_1 m_2$	s
~~m_1~~ m_2	$s m_1$
$m_2 m_4 m_3$	$s m_1$
~~m_2~~ $m_4 m_3$	$s m_1 m_2$
$m_4 m_3 m_3 m_5$	$s m_1 m_2$
~~m_4~~ $m_3 m_3 m_5$	$s m_1 m_2 m_4$
$m_3 m_3 m_5 m_6 r$	$s m_1 m_2 m_4$



The solution path is $s \rightarrow m_1 \rightarrow m_4 \rightarrow r$.

Example:



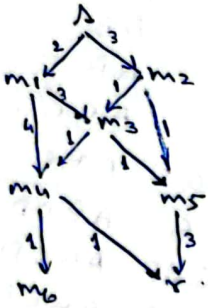


OPEN	CLOSED.
A.	
~~A~~	A.
BC.	A.
~~B~~C	AB.
CDE	AB.
~~C~~DE	ABC.
DEDG.	ABC.
~~D~~EDG	ABCD
EDG~~D~~F	ABCD
~~E~~DGEF	ABCDE.
~~D~~GEF	ABCD.
GEFBF	ABCD.

Note:

1. OPEN is the list of nodes generated in the explicit graph but not yet expanded. (just like dfs).
2. CLOSED is the list of expanded nodes in the graph (and is checking for duplicate nodes).
3. The idea of cleanup is not applied here because they are not considering the paths.
4. There are b^d nodes in a tree having depth d and branching factor b. In the worst case, we are to store all the b^d nodes so overhead is high compared to dfs.
5. Dfs is recursive and at the time of expanding it basically keeps the path in implicit stack but in case of bfs it is relatively difficult to implement in recursive manner because it switches from one path to another.
6. Solution path is always optimal in terms depth of a node but may not be optimal in terms of cost.
7. Duplicate nodes are not expanded but copies of the same node may appear in OPEN.
8. Advantages are.
   (i) Shallowed goal guaranteed - may not be optimal in terms of cost in case of graphs.
   (ii) Faster solution in certain graphs.
9. Disadvantages are
   (i) Higher memory requirement
   (ii) More housekeeping work, more overhead and relatively difficult to implement
   (iii) Not suited to human's problem solving approach.

Best first search: In general, dfs and bfs are blind methods since they take a node and then expand. These algorithms are not choosing the node for expansion so that it will be on the solution path.



Here the solutions for dfs is $s \to m_1 \to m_4 \to r$ where the cost is 7 units and for bfs is $s \to m_2 \to m_3 \to m_4 \to r$ where the cost is 6 units.

We seek to determine a path of least cost from the root node s to the goal node r.

Let $g(n)$ be the cost of currently known best path from s to r and $g^*(n)$ is the cost of the best path from s to r. Obviously $g^*(n) \leq g(n)$. $f(s)$ is the optimal solution cost.

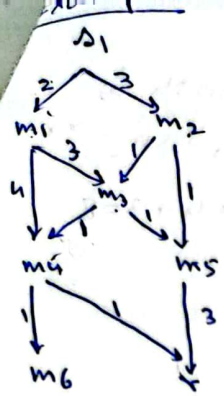
Algorithm Uniform Cost (U):

Input: root/start node, goal node and method of expansion (operation)

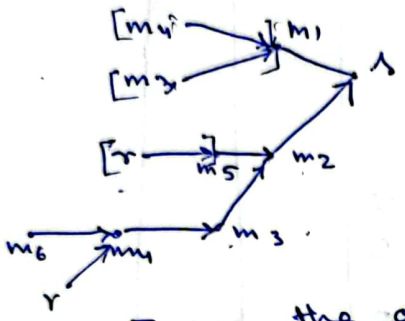
Output: optimal solution path.

OPEN and CLOSED are initially empty.

```
begin
U1: g(s) = 0; put s in OPEN; found = false;
    while OPEN not empty and not(found) do
    begin
U2:   select a node n from OPEN with minimum g value;
      /* resolve ties arbitrarily but in favour of a goal node */
      remove n from OPEN and put it in CLOSED;
      if n is a goal node then found = true else
      begin
          expand n generating all its immediate successors
          for each immediate successor ni of n do
          begin
              g = g(n) + c(n, ni);
              if ni is not already in OPEN or CLOSED then
              begin
                  g(ni) = g; put ni in OPEN;
                  direct backward pointer from ni to n;
              end
              else if ni in OPEN and g(ni) > g then
              begin
                  g(ni) = g;
                  redirect backward pointer from ni to n;
              end;
          end;
      end;
end;
if found then output g(n) and solution path by tracing back
through pointers
else output failure message;
```



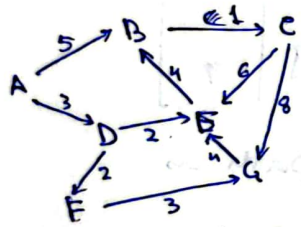
Instant	s	m_1	m_2	m_3	m_4	m_5	m_6	r.
0	0							
1	[0]	2	3					
2		[2]	3	5	6			
3			[3]	4	6	4		
4				[4]	5	5		
5					5	[5]		8
6					[5]		6	6
7								[6]



Optimal solution cost is 6 units.
optimal solution path is

$s \rightarrow m_2 \rightarrow m_3 \rightarrow m_4 \rightarrow r$.

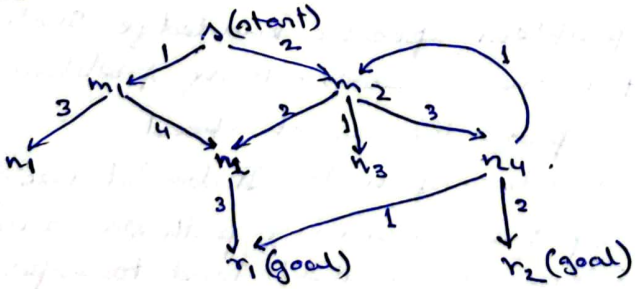
Problem: Trace the optimal solution path for the following graph.



Optimal cost is 8.
optimal path.
$A \rightarrow D \rightarrow F \rightarrow G$.

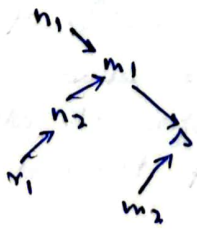
Instant	A	B	C	D	E	F	G
0	0						
1	[0]	5		3			
2		5		[3]	5	5	
3		[5]	6.		5	5	
4			6		[5]	5	
5			6			[5]	8
6			[6]				8
7							[8]

Problem: Trace dfs, bfs and uc. for the following graph.



Using dfs:

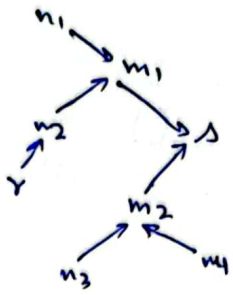
OPEN	CLOSED
s	
$m_1 m_2$	s
$n_1 n_2 m_2$	$s m_1$
$r_1 m_2$	$s m_1 m_2$.



solution path is $s \rightarrow m_1 \rightarrow n_2 \rightarrow r$.
cost = 1+4+3 = 8.

Using bfs:

OPEN	CLOSED
s	
$m_1\ m_2$	s
$m_2\ n_1\ n_2$	$s\ m_1$
$n_2\ n_3\ m_4$	$s\ m_1\ m_2$
$n_4\ r_1$	$s\ m_1\ m_2\ n_2$



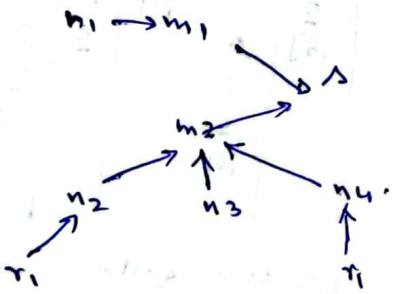
solution path is

$s \rightarrow m_1 \rightarrow n_2 \rightarrow r$

cost = 1+4+3=8.

Using uniform cost search:

Instant	s	m_1	m_2	n_1	n_2	n_3	n_4	r_1	r_2
0	0								
1	[0]	1	2						
2		[1]	2	4	5				
3			[2]	4	4	3	5		
4				4	4	[3]	5		
5				[4]	4		5		
6					[4]		5	7	
7							[5]	6	7
8								[6]	7



Solution path is

$s \rightarrow m_2 \rightarrow n_4 \rightarrow r_1$.

cost = 2+3+1=6.

Note:

1. Uniform cost search lack problem specific knowledge. Such methods are prohibitively inefficient in many cases. Using problem-specific knowledge can dramatically improve the search speed.
2. Uniform cost works with $g(n)$ values of nodes. It does not use any information about the cost of the remaining path from a node to a goal node. So in many problems, this method tends to expand too many nodes.
3. It is possible to cut down the number of node expansions by using the estimate of the cost of the remaining path from a node to a goal node. Heuristic search approach attempts to direct the search by using heuristic estimates of the cost of the path from a node to a goal node.
4. several approaches to heuristic search
   (a) A* approach (b) marketing approach (c) propagation of values (d) recursive approach.

Heuristic means "rule of thumb". Heuristics are criteria, methods or principles for deciding which among several alternative courses of action promises to be most efficient in order to achieve some goal.

A heuristic function ~~is defined~~ at node n, denoted by $h(n)$, is an estimate of the optimum cost from the current node to a goal.

Eg1: we want a path from Kolkata to Bolpur. The heuristic for Bolpur may be straight-line distance between Kolkata and Bolpur.

$h(\text{Kolkata}) = \text{Euclidean Distance}(\text{Kolkata}, \text{Bolpur})$.

Eg2(a) Misplaced tiles heuristics is the no of tiles out of place.

2	8	3
1	6	4
	7	5

Initial.

1	2	3
8		4
7	6	5

Goal.

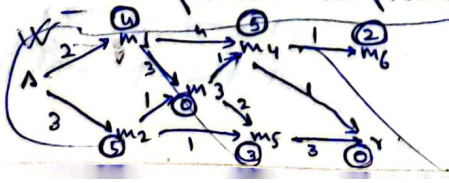
$h(n) = 5$ because the tiles 2, 8, 1, 6 and 7 are out of place.

(b) Another heuristic is the Manhattan distance heuristic which is the sum of the distance of the tiles that are out of place. The distance of a tile is measured by the sum of the differences in the x-positions and in the y-positions.

$\therefore h(n) = 1+1+0+0+0+1+1+2 = 6$.

Heuristic estimates:

1. Non-negative heuristic estimates $h(n)$ associated with each node in a graph G, here $h(n)$ is an estimate of the cost of a minimum cost path from n to a goal node; $h(n)$ is estimated from the problem domain.
2. $f(n) = g(n) + h(n)$; $f(n)$ gives an estimate of the cost of a minimum cost path from the start node to a goal node which is a constraint to pass through n.
3. $h^*(n)$ is the actual cost of a minimal cost path from n to a node, infinite if no path exists; $g^*(n)$ is the actual cost of minimal cost path from s to n; $f^*(n) = g^*(n) + h^*(n)$ is the actual cost of a minimal cost path from s to a goal node which is contraint to pass through n.
4. Assumptions about $h(n)$:
   (a) $h(m) \geq 0$ for every node m.
   (b) $h(s) = 0$ and $h(r) \geq 0$
   (c) if m lies on a solution path then $h(m)$ is finite otherwise $h(m)$ is finite or infinite.



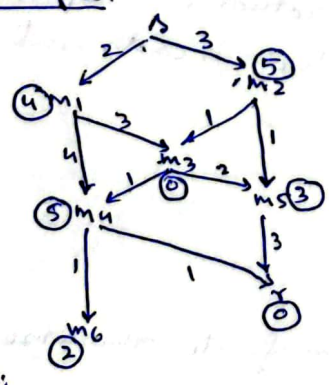
$g^*(m_1) = 2$
$g^*(m_2) = 3$
$g^*(m_3) = 4$
$g^*(m_4) = 5$
$g^*(m_5) = 4$
$g^*(m_6) = 6$
$g^*(r) = 6$

$h^*(m_1) = 5$
$h^*(m_2) = 3$
$h^*(m_3) = 2$
$h^*(m_4) = 1$
$h^*(m_5) = 3$
$h^*(m_6) = \infty$

Algorithm A*:

```
begin
A1: g(s)=0; f(s)=0; found = false;
    put s in OPEN;
    while OPEN is not empty and not(found) do
    begin
A2: select a node n from OPEN with minimum f-value
    /* resolve ties arbitrarily but in favour of a goal node */.
    Remove n from OPEN and put n in CLOSED.
    if n is a goal node then found=true else
    begin
        expand n generating all its successors if any;
        for each immediate successor ni of n do
        begin
            g=g(n)+c(n,ni);
            if ni is not in OPEN or CLOSED then.
            begin
                g(ni)=g; f(ni)=g+h(ni);
                put ni in OPEN;
                direct backward pointer from ni to n;
            end
            else if g(ni)>g then
            begin
                g(ni)=g; f(ni)=g+h(ni)
                redirect backward pointer from ni to n;
                if ni is in CLOSED then remove ni from CLOSED
                                        and put it in OPEN;
            end;
        end end
    end;
    if found then output f(n) and solution path by tracing back
        through pointers;
    else output failure message;
end;
```

Example:



Instant	s	m_1	m_2	m_3	m_4	m_5	m_6	r
0	0+0							
1	0+0	2+4	3+5					
2		2+4	3+5	5+0	6+5			
3			3+5	5+0	6+5	7+3		
4			3+5	4+0	6+5	4+3		
5				4+0	5+5	4+3		
6					5+5	4+3		7+0
7								7+0

Go:

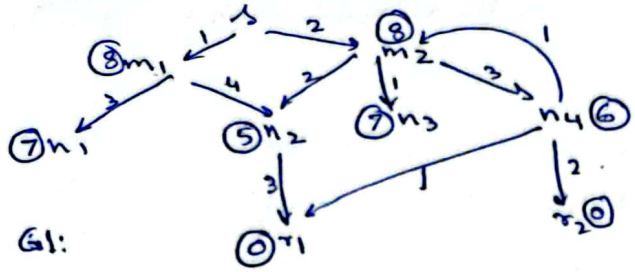
Solution is f(n) = 7

Nodes expanded are $s \to m_1 \to m_3 \to m_2 \to m_3 \to m_5 \to r$.

Solution path is $s \to m_2 \to m_5 \to r$. This is not the optimal path

The node m_3 is expanded more than once.

Example:



G1:

Instant	s	m_1	m_2	M	n_2	n_3	n_4	r_1	r_2
0	0+0								
1	0+0	1+8	2+8						
2		1+8	2+8	4+7	5+5				
3			2+8	4+7	4+5	3+7	5+6		
4				4+7	4+5	3+7	5+6	7+0	
5								7+0	

Solution path is $s \to m_2 \to n_2 \to r_1$.
Cost is 7 units.

Repeating from instant 2 (alternative).

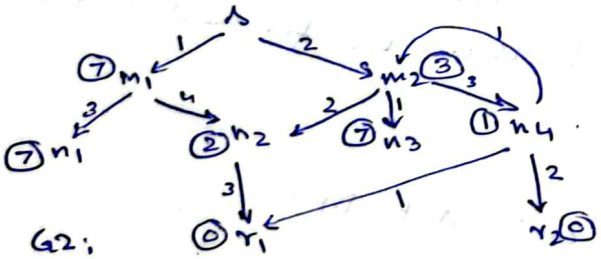
2		1+8	2+8	4+7	5+5				
3			2+8	4+7	5+5			8+0	
4								8+0	

Solution path is $s \to m_1 \to n_2 \to r_1$
Cost is 8 units.

The optimal cost is 6 when the solution path is $s \to m_2 \to n_4 \to r$.

Note: If h(n) is zero for all nodes then A* is reduced to uniform cost and optimal cost is guaranteed. So we can say that if the heuristics are not done properly we may not get the optimal path.

Example:



G2:

Instant	s	m_1	m_2	n_1	n_2	n_3	n_4	r_1	r_2
0	0+0								
1	0+0	1+7	2+3						
2		1+7	2+3						
3		1+7			4+2	3+7	5+1		
4					4+2	3+7	5+1	7+0	
5						3+7	5+1	6+0	7+0
6								6+0	

Solution path is $s \to m_2 \to n_4 \to r_1$.
Cost is 6 units.

A heuristic h is admissible if for each node $n \in G$, $0 \leq h(n) \leq h^*(n)$, otherwise it is called inadmissible.

A* search properties:

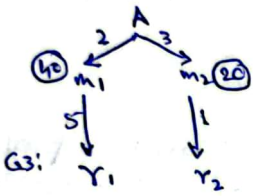
(i) The algorithm A* is admissible, i.e, provided a solution exists, the first solution found by A* is an optimal solution ie, A* finds optimal solution path if heuristic estimates are admissible.

(ii) A* is also complete.

**

Note: In G0 and G1, heuristics were inadmissible. If we now change the heuristics of G1 to that in G2 we get correct solution.

In G3, eventhough heuristics are inadmissible, it gives optimal solution and hence the reverse statement (that algorithm A* will not find optimal solution path if heuristic estimates are inadmissible) is not true.



** (iii) A* is optimally efficient for a given heuristic - of the optimal search algorithms that expands search paths from the root node, it can be shown that no other optimal algorithm will expand fewer nodes and find a solution. However, the no. of nodes searched is still exponential in the worst case.

(iv) A* must keep all nodes it is considering in memory.

(v) A* is much more efficient than uninformed methods.

Limitations of A*

(i) Worst case performance is poor.

(ii) Size of memory is appreciable in contrast to DFS.

(iii) Repeated node expansion.

(iv) In case of inadmissible heuristic no idea about the quality of solution.

(v) Overhead of ~~open~~ OPEN just as BFS.

(vi) OPEN must be implemented as a priority queue.

Advantages of A*

(i) A* is quite fast on the average and is widely used.

(ii) A* never stores the entire explicit graphs and hence make repeated expansion.

Hill Climbing (or gradient ascent / descent)

- Local search algorithm with greedy approach and no backtracking

Step 1: Evaluate the initial state.

" 2: Loop until a solution is found or there are no new operators left

" 3: Select and apply new operator.

" 4: Evaluate new state

(i) if it is a goal state the quit.

(ii) if it is better than current state then make it new current state.

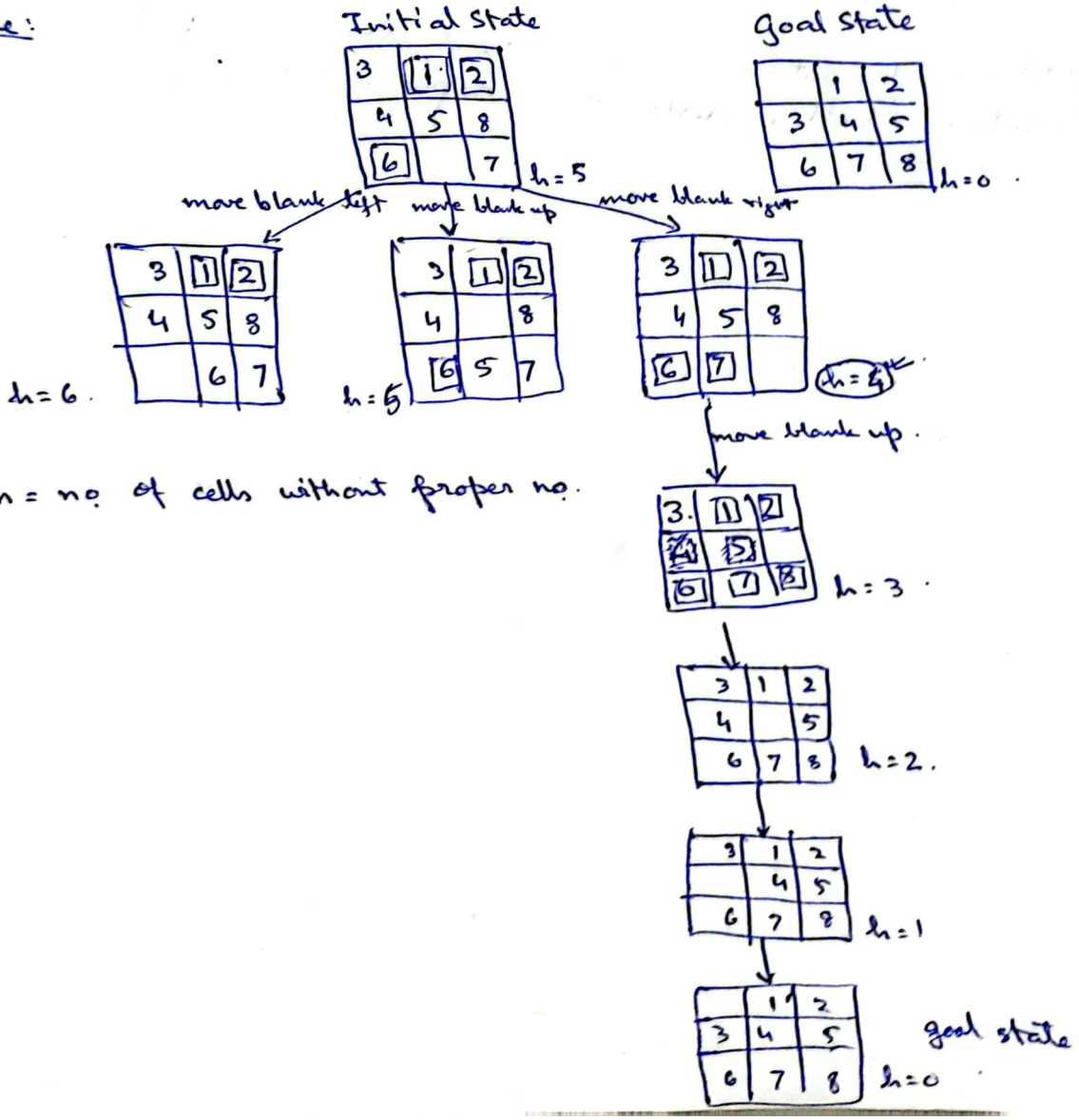
(iii) if it is not better than current state then goto step 2.

Note:
Here we iteratively maximize the "value" of the current state by replacing it by successor state that has highest value as long as possible.

No search tree is maintained.

Only the states directly reachable from the current state are considered.

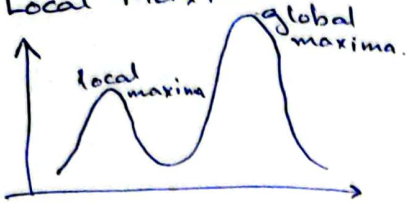
Example:



h = no of cells without proper no.

Limitations of Hill climbing problem:

1. Local Maxima:

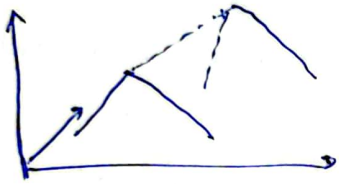


One the top of a hill is reached the algorithm will halt since every possible step leads down.

2. Plateax:

If the landscape is flat, meany ~~so~~ many states have the same goodness, the algorithm degenerates to a random walk.

3. Ridges:



If the landscape contains ridges, local improvements may follow a zigzag up the ridge, slowing down the search.

Gradient descent is an inverted version of hill-climbing in which better states are represented by lower cost values.