Shri Vile Parle Kelavani Mandal's

INSTITUTE OF TECHNOLOGY

DHULE (M.S.)

DEPARMENT OF COMPUTER ENGINEERING

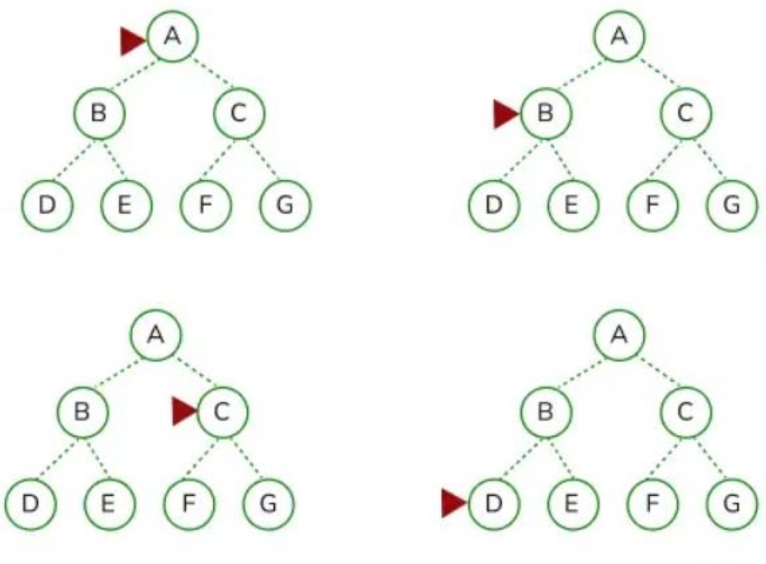
Subject : Artificial Intelligence Lab			Remark
Name :		**Roll No. :**	
Class : SY Btech	**Batch :**	**Division:**	
Expt. No. : 04	**Date :**		Signature
Title : Solve any problem using breadth first search.			

Theory :

The Breadth-First Search is a traversing algorithm used to satisfy a given property by searching the tree or graph data structure. It belongs to uninformed or blind search AI algorithms as it operates solely based on the connectivity of nodes and doesn't prioritize any particular path over another based on heuristic knowledge or domain-specific information.

It doesn't incorporate any additional information beyond the structure of the search space. It is optimal for unweighted graphs and is particularly suitable when all actions have the same cost. Due to its systematic search strategy, BFS can efficiently explore even infinite state spaces.

BFS Working



- Originally it starts at the root node, then it expands to all of its successors It systematically explores all its neighbouring nodes before moving to the next level of nodes. (As shown in the above image, it starts from the root node A and then expands its successor B)
- This process of extending to the root node's immediate neighbours, then to their neighbours, and so on, lasts until all the nodes within the graph have been visited or until the specific condition is met. From the above image we can observe that after visiting the node B it moves to node C. When level 1 is completed, it further moves to the next level i.e 2 and explores node D. Then it will move systematically to node E, node F and node G. After visiting node G, it will terminate.

Code:

```
% --- Facts: Graph edges ---
edge(a, b).
edge(a, c).
edge(b, d).
edge(b, e).
edge(c, f).
edge(c, g).

% --- BFS driver ---
bfs(Start, Goal, Path) :-
   bfs_search([[Start]], Goal, RevPath),
   reverse(RevPath, Path).

% --- BFS search loop ---
bfs_search([[Goal|RestPath] | _], Goal, [Goal|RestPath]). % Goal found

bfs_search([[Node|RestPath] | OtherPaths], Goal, Path) :-
   findall([Next,Node|RestPath],
          (edge(Node, Next), \+ member(Next, [Node|RestPath])),
          NewPaths),
   append(OtherPaths, NewPaths, UpdatedQueue),
   bfs_search(UpdatedQueue, Goal, Path).
```

Output:

```
bfs(a,g,Path).
Path = [a, c, g]
```

Shri Vile Parle Kelavani Mandal's

INSTITUTE OF TECHNOLOGY

DHULE (M.S.)

DEPARMENT OF COMPUTER ENGINEERING

Subject : Artificial Intelligence Lab	Remark
Name : **Roll No. :**	
Class : SY Btech **Batch :** **Division:**	Signature
Expt. No. : 05 **Date :**	
Title : Solve an 8-puzzle problem using the best first search.	

Theory:

Best First Search (Informed Search)

Best First Search is a heuristic search algorithm that selects the most promising node for expansion based on an evaluation function. It prioritizes nodes in the search space using a heuristic to estimate their potential. By iteratively choosing the most promising node, it aims to efficiently navigate towards the goal state, making it particularly effective for optimization problems.

We are given an edge list of a graph where every edge is represented as (u, v, w). Here u, v and w represent source, destination and weight of the edges respectively. We need to do Best First Search of the graph (Pick the minimum cost edge next).

Examples:

Input: source = 0, target = 9
edgeList = [[0, 1, 3], [0, 2, 6], [0, 3, 5], [1, 4, 9],
[1, 5, 8], [2, 6, 12], [2, 7, 14], [3, 8, 7],
[8, 9, 5], [8, 10, 6], [9, 11, 1], [9, 12, 10], [9, 13, 2]]
Output: 0 1 3 2 8 9

Explanation: Following describes the working of best first search:

- Start at node 0. Among its neighbors, node 1 has the lowest cost (edge cost 3), so move from 0 to 1.
- From node 1, since a better path forward isn't available, backtrack to node 0.

- From node 0, choose the next best neighbor—node 3 (edge cost 5)—and move to node 3.
- At node 3, find that moving to node 2 leads to a lower incremental cost.
- Then proceed from node 2 to node 8, and finally from node 8 to node 9 (the target).

Input: source = 0, target = 8
edgeList = [[0, 1, 3], [0, 2, 6], [0, 3, 5], [1, 4, 9],
[1, 5, 8], [2, 6, 12], [2, 7, 14], [3, 8, 7],
[8, 9, 5], [8, 10, 6], [9, 11, 1], [9, 12, 10], [9, 13, 2]]
Output: 0 1 3 2 8
Explanation: Following describes the working of best first search:

- Start at node 0. Among its neighbors, node 1 has the lowest cost (edge cost 3), so move from 0 to 1.
- From node 1, since a better path forward isn't available, backtrack to node 0.
- From node 0, choose the next best neighbor—node 3 (edge cost 5)—and move to node 3.
- At node 3, find that moving to node 2 leads to a lower incremental cost.
- Then proceed from node 2 to node 8 (the target)

Code:

```
% best_first_8puzzle.pl
% Greedy Best-First Search for 8-puzzle
% State representation: list of 9 numbers, 0 = blank.
% Example goal: [1,2,3,4,5,6,7,8,0]

%% Public entry
solve_best_first(Start, Goal, Path) :-
   h_misplaced(Start, Goal, H0),
   best_first([H0-(Start-[Start])], Goal, [], Path).  % open: list of H-(State-PathRev)

%% Main loop
best_first([], _, _, _) :-                     % no solution
   fail.
best_first([_H-(State-PathRev)|_], Goal, _, Path) :-
   State == Goal,                          % goal test
   reverse(PathRev, Path), !.                  % return path (start..goal)
best_first([_H-(State-PathRev)|RestOpen], Goal, Closed, Path) :-
   findall(Hn-(Next-NewPathRev),
         ( move(State, Next),
           \+ member(Next, PathRev),        % avoid cycles by path check
           \+ member(Next, Closed),        % optional: avoid states already closed
           h_misplaced(Next, Goal, Hn),
           NewPathRev = [Next|PathRev]
```

```
            ),
            Children),
    append(RestOpen, Children, Open2),
    keysort(Open2, SortedOpen),                 % sort by heuristic (smallest H first)
    best_first(SortedOpen, Goal, [State|Closed], Path).

%% Moves: swap blank (0) with an adjacent tile (using indices 1..9).
move(State, NextState) :-
    nth1(BlankPos, State, 0),
    adj(BlankPos, SwapPos),
    swap(State, BlankPos, SwapPos, NextState).

%% adjacency in 3x3 grid (symmetric)
adj(1,2). adj(1,4).
adj(2,1). adj(2,3). adj(2,5).
adj(3,2). adj(3,6).
adj(4,1). adj(4,5). adj(4,7).
adj(5,2). adj(5,4). adj(5,6). adj(5,8).
adj(6,3). adj(6,5). adj(6,9).
adj(7,4). adj(7,8).
adj(8,5). adj(8,7). adj(8,9).
adj(9,6). adj(9,8).

%% swap elements at positions I and J (1-based)
swap(State, I, J, NewState) :-
    nth1(I, State, EI),
    nth1(J, State, EJ),
    replace(State, I, EJ, Temp),
    replace(Temp, J, EI, NewState).

replace([_|T], 1, X, [X|T]).
replace([H|T], I, X, [H|R]) :-
    I > 1, I1 is I - 1,
    replace(T, I1, X, R).

%% Heuristic: number of misplaced tiles (excluding blank)
h_misplaced(State, Goal, H) :-
    h_misplaced_acc(State, Goal, 0, H).

h_misplaced_acc([], [], Acc, Acc).
h_misplaced_acc([0|Ts], [_|Gs], Acc, H) :-   % ignore blank
    !, h_misplaced_acc(Ts, Gs, Acc, H).
h_misplaced_acc([X|Ts], [Y|Gs], Acc, H) :-
    X == Y, !,
    h_misplaced_acc(Ts, Gs, Acc, H).
h_misplaced_acc([_|Ts], [_|Gs], Acc, H) :-
```

```
    Acc1 is Acc + 1,
    h_misplaced_acc(Ts, Gs, Acc1, H).
```

Output:

```
solve_best_first([1,2,3,4,5,6,7,0,8],[1,2,3,4,5,6,7,8,0],Path).

Path = [[1, 2, 3, 4, 5, 6, 7, 0|...], [1, 2, 3, 4, 5, 6, 7|...]].
```

Shri Vile Parle Kelavani Mandal's

INSTITUTE OF TECHNOLOGY

DHULE (M.S.)

DEPARMENT OF COMPUTER ENGINEERING

Subject : Artificial Intelligence Lab			Remark
Name :		**Roll No. :**	
Class : SY Btech	**Batch :**	**Division:**	Signature
Expt. No. : 07	**Date :**		
Title : Solve travelling salesman problem.			

Theory:

The travelling salesman problem is a graph computational problem where the salesman needs to visit all cities (represented using nodes in a graph) in a list just once and the distances (represented using edges in the graph) between all these cities are known.
The solution that is needed to be found for this problem is the shortest possible route in which the salesman visits all the cities and returns to the origin city.

Code:

```
% ----- Facts -----

% distance(City1, City2, Distance).

distance(a, b, 10).

distance(a, c, 15).

distance(a, d, 20).

distance(b, c, 35).

distance(b, d, 25).
distance(c, d, 30).
% distance is symmetric
dist(X, Y, D) :- distance(X, Y, D).
dist(X, Y, D) :- distance(Y, X, D).

% ----- Generate tour -----
% find all permutations of cities (except start), calculate cost
tsp(Start, Path, Cost) :-
    findall(C, (distance(Start, C, _)), Cities1),
    sort(Cities1, Cities),              % get unique cities
    permutation(Cities, Perm),
    append([Start|Perm], [Start], Path), % full cycle
    path_cost(Path, Cost).

% ----- Calculate path cost -----
path_cost([_], 0).
path_cost([A,B|T], Cost) :-
    dist(A, B, D),
    path_cost([B|T], Rest),
    Cost is D + Rest.

% ----- Find best path -----
tsp_best(Start, BestPath, MinCost) :-
    setof((C, P), tsp(Start, P, C), [(MinCost, BestPath)|_]).
```

Output:

```
tsp_best(a,Path,Cost).
Path = [a, b, d, c, a],
```

Cost = 80.