**19I510 Design and Analysis of Algorithms**

**Exercise10 – Branch and Bound**

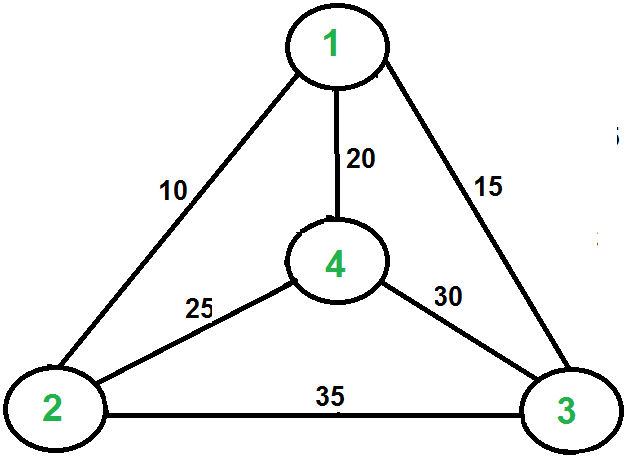
**Branch and bound is an algorithm design paradigm which is generally used for solving combinatorial optimization problems. These problems typically exponential in terms of time complexity and may require exploring all possible permutations in worst case. Branch and Bound solve these problems relatively quickly.**

**Exercise:**

1. Traveling Salesman Problem
2. Job Assignment Problem
3. Implementation of 0/1 Knapsack
4. 8 puzzle Problem
5. **Implementation of Traveling Salesman Problem**

Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

For example, consider the graph shown in figure. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is 10+25+30+15 which is 80.



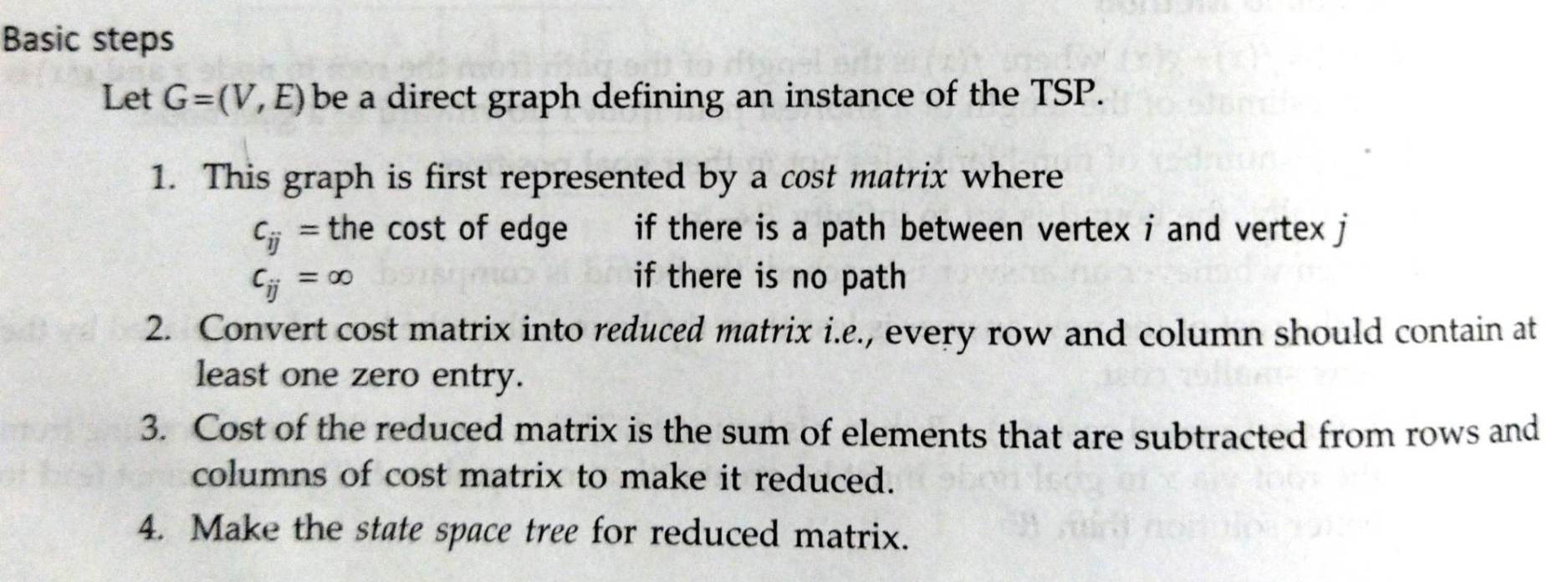
The problem is a famous [NP hard](http://www.geeksforgeeks.org/np-completeness-set-1/)problem. There is no polynomial time know solution for this problem.

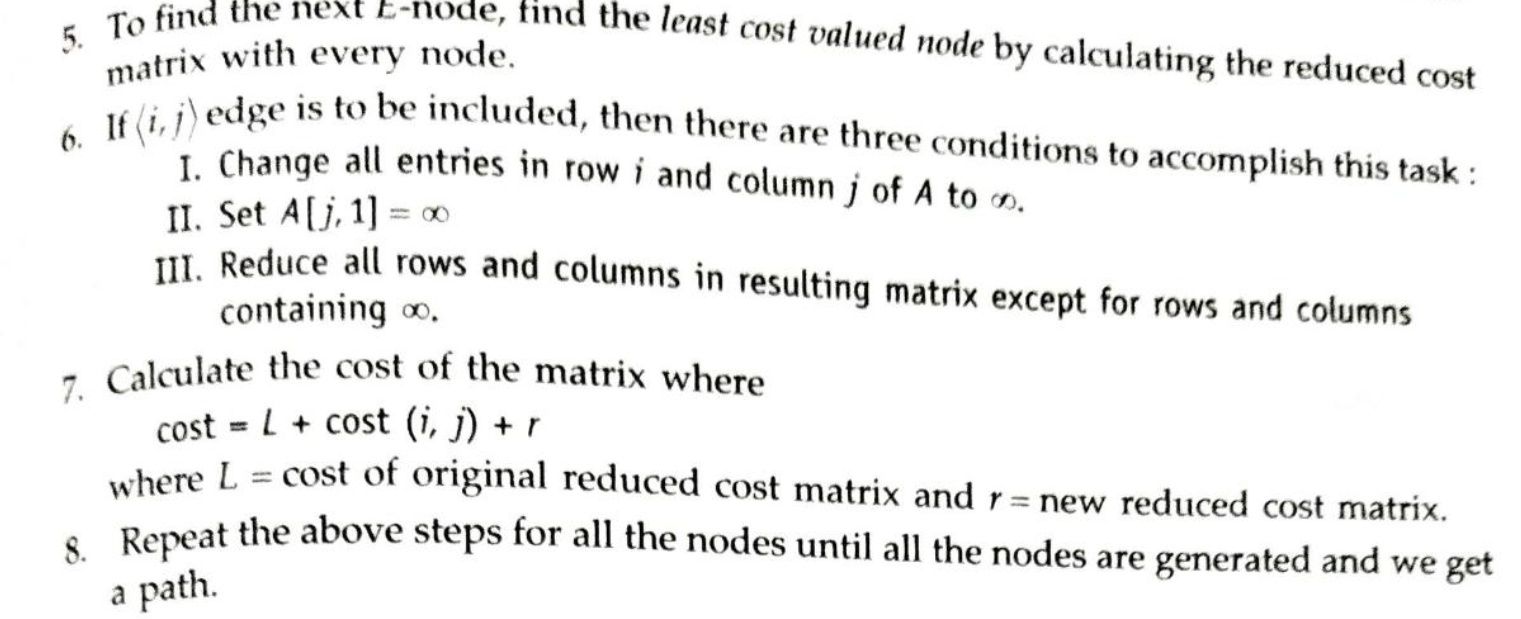
There are different solutions available for the travelling salesman problem

Naïve solution and Dynamic programming gives almost O(n!)

Approximate solution using Minimum spanning tree

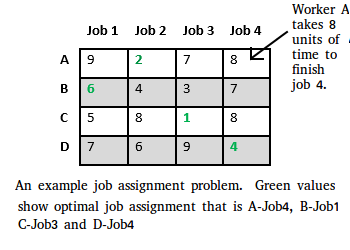
**Refer DAA Theory PPT for solving Problem**





1. Implementation of Job Assignment Problem

Let there be N workers and N jobs. Any worker can be assigned to perform any job, incurring some cost that may vary depending on the work-job assignment. It is required to perform all jobs by assigning exactly one worker to each job and exactly one job to each agent in such a way that the total cost of the assignment is minimized.

[](http://www.geeksforgeeks.org/wp-content/uploads/jobassignment.png)

Possible approaches for the problem

**Solution 1: Brute Force**

complexity is O(n!).

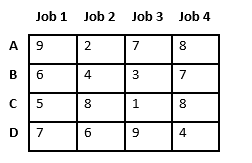
**Solution 2:**[**Hungarian Algorithm**](http://www.geeksforgeeks.org/hungarian-algorithm-assignment-problem-set-1-introduction/)

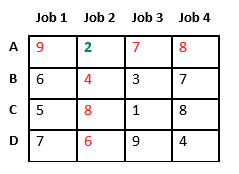
complexity of O(n^3).

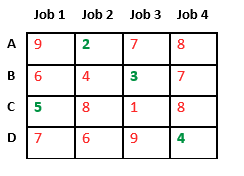
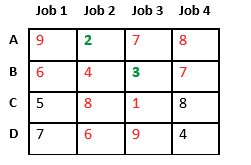
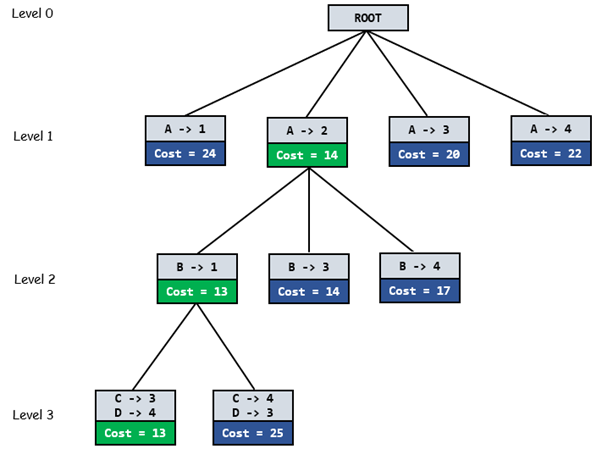
**Optimal Solution using Branch and Bound**

There are two approaches to calculate the cost function:

1. For each worker, we choose job with minimum cost from list of unassigned jobs (take minimum entry from each row).
2. For each job, we choose a worker with lowest cost for that job from list of unassigned workers (take minimum entry from each column).

Let’s take below example and try to calculate promising cost when Job 2 is assigned to worker A.  
[](http://www.geeksforgeeks.org/wp-content/uploads/jobassignment2.png)  
Since Job 2 is assigned to worker A (marked in green), cost becomes 2 and Job 2 and worker A becomes unavailable (marked in red).

[](http://www.geeksforgeeks.org/wp-content/uploads/jobassignment3.png)  
Now we assign job 3 to worker B as it has minimum cost from list of unassigned jobs. Cost becomes 2 + 3 = 5 and Job 3 and worker B also becomes unavailable.

[](http://www.geeksforgeeks.org/wp-content/uploads/jobassignment5.png)[](http://www.geeksforgeeks.org/wp-content/uploads/jobassignment4.png)  
Finally, job 1 gets assigned to worker C as it has minimum cost among unassigned jobs and job 4 gets assigned to worker C as it is only Job left. Total cost becomes 2 + 3 + 5 + 4 = 14.  
 

**Algorithm**:

\* findMinCost uses Least() and Add() to maintain the

list of live nodes

Least() finds a live node with least cost, deletes

it from the list and returns it

Add(x) calculates cost of x and adds it to the list

of live nodes

Implements list of live nodes as a min heap \*/

// Search Space Tree Node

node

{

int job\_number;

int worker\_number;

node parent;

int cost;

}

// Input: Cost Matrix of Job Assignment problem

// Output: Optimal cost and Assignment of Jobs

algorithm findMinCost (costMatrix mat[][])

{

// Initialize list of live nodes(min-Heap)

// with root of search tree i.e. a Dummy node

while (true)

{

// Find a live node with least estimated cost

E = Least();

// The found node is deleted from the list

// of live nodes

if (E is a leaf node)

{

printSolution();

return;

}

for each child x of E

{

Add(x); // Add x to list of live nodes;

x->parent = E; // Pointer for path to root

}

}

}

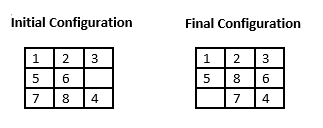
1. **Implementation of 0/1 Knapsack**

**Algorithm:**

1. Sort all items in decreasing order of ratio of value per unit weight so that an upper bound can be computed using Greedy Approach.
2. Initialize maximum profit, maxProfit = 0
3. Create an empty queue, Q.
4. Create a dummy node of decision tree and enqueue it to Q. Profit and weight of dummy node are 0.
5. Do following while Q is not empty.
   * Extract an item from Q. Let the extracted item be u.
   * Compute profit of next level node. If the profit is more than maxProfit, then update maxProfit.
   * Compute bound of next level node. If bound is more than maxProfit, then add next level node to Q.
   * Consider the case when next level node is not considered as part of solution and add a node to queue with level as next, but weight and profit without considering next level nodes.
6. **Implementation of 8 puzzle Problem**

Given a 3×3 board with 8 tiles (every tile has one number from 1 to 8) and one empty space. The objective is to place the numbers on tiles to match final configuration using the empty space. We can slide four adjacent (left, right, above and below) tiles into the empty space.

For example,

[](http://www.geeksforgeeks.org/wp-content/uploads/8puzzle2.png)

Approaches

Brute Force : Either DFS or BFS

**Branch and Bound**

Assume that moving one tile in any direction will have 1 unit cost. Cost function defined for 8-puzzle algorithm as below:

c(x) = f(x) + h(x) where

f(x) is the length of the path from root to x (the number of moves so far) and

h(x) is the number of non-blank tiles not in their goal position (the number of mis-placed tiles). There are at least h(x) moves to transform state x to a goal state

Pseudo code:

/\* Algorithm LCSearch uses c(x) to find an answer node

\* LCSearch uses Least() and Add() to maintain the list

of live nodes

\* Least() finds a live node with least c(x), deletes

it from the list and returns it

\* Add(x) adds x to the list of live nodes

\* Implement list of live nodes as a min heap \*/

struct list\_node

{

list\_node \*next;

// Helps in tracing path when answer is found

list\_node \*parent;

float cost;

}

algorithm LCSearch(list\_node \*t)

{

// Search t for an answer node

// Input: Root node of tree t

// Output: Path from answer node to root

if (\*t is an answer node)

{

print(\*t);

return;

}

E = t; // E-node

Initialize the list of live nodes to be empty;

while (true)

{

for each child x of E

{

if x is an answer node

{

print the path from x to t;

return;

}

Add (x); // Add x to list of live nodes;

x->parent = E; // Pointer for path to root

}

if there are no more live nodes

{

print ("No answer node");

return;

}

// Find a live node with least estimated cost

E = Least();

// The found node is deleted from the list of

// live nodes

}

}