Greedy Method

Greedy method is another important algorithm design paradigm. Before going in detail about the method let us see what an optimization problem is about.

For an optimization problem, we are given a set of *constraints* and an *optimization function*. Solutions that satisfy the constraints are called *feasible solutions*. And feasible solution for which the optimization function has the best possible value is called an *optimal solution*.

In a *greedy method* we attempt to construct an optimal solution in stages. At each stage we make a decision that appears to be the best (under some criterion) at the time. A decision made at one stage is not changed in a later stage, so each decision should assure feasibility. Greedy algorithms do not always yield a genuinely optimal solution. In such cases the greedy method is frequently the basis of a *heuristic approach*. Even for problems which can be solved exactly by a greedy algorithm, establishing the correctness of the method may be a non-trivial process.

In general, greedy algorithms have five pillars:

1. A candidate set, from which a solution is created
2. A selection function, which chooses the best candidate to be added to the solution
3. A feasibility function, that is used to determine if a candidate can be used to contribute to a solution
4. An objective function, which assigns a value to a solution, or a partial solution
5. A solution function, which will indicate when we have discovered a complete solution

Control Abstraction

```
SolutionType Greedy (type a[], int n)
{
    SolutionType solution = EMPTY;

    for(int i=1; i<=n; i++)
    {
            Type x = Select(a);
            If Feasible(solution, x)
                    solution =  Union (solution, x);
    }
    return solution;
}
```

Select – select an input from a[] and removes it from the array
Feasible – check feasibility
Union - combines x with solution and updates objective function

3.1 Knapsack problem

Problem: Given n inputs and a knapsack or bag. Each object i is associated with a weight w_i and profit p_i. If fraction of an object x_i, $0 \le x_i \ge 1$ is placed in knapsack earns profit $p_i x_i$. Fill the knapsack with maximum profit.

$$\text{Maximize} \sum_{1 \le i \le n} p_i x_i$$

$$\text{Subject to} \sum_{1 \le i \le n} w_i x_i \le m \quad \text{and} \quad 0 \le x_i \le 1,\ 0 \le i \le n$$

- If sum of all weights ≤ m, then all $x_i = 1$, $0 \le x_i \le 1$, is an optimal solution
- If sum >m, all x_i cannot be equal to 1. Then optimal solution fills knapsack exactly

Example 3.1

Number of inputs n = 3

Size of the knapsack m = 20

(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum p_i x_i$
(1, 2/15, 0)	20	28.2
(0, 2/3, 1)	20	31
(0, 1, 1/2)	20	31.5

Algorithm 3.1

```
void GreedyKnapsack (float m, int n)
{
    // p[i]/w[i] ≥ p[i+1]/w[i+1]
    for (int i=1; i<=n; i++)
            x[i] = 0.0;
    float u = m;
    for (i=1; i<=n; i++)
    {
            if(w[i]  > u)  break;
            x[i] = 1.0;
            u = u – w[i];
    }
    if (i<=m)  x[i] = u/w[i];
}
```

Theorem 3.1

If objects are selected in order of decreasing p_i/w_i, then knapsack finds an optimal solution

Proof

Given that $p_1/w_1 \ge p_2/w_2 \ge \ldots\ldots p_n/w_n$

Let $X = (x_1, x_2, \ldots, x_n)$ be the solution generated by the Greedy algorithm

Let j be the smallest index such that $x_j < 1$.

Then we know that

$x_i = 1$, when $i < j$
$x_i = 0$, when $i > j$
$0 \le x_i \le 1$, when $i = j$

and that $\sum_{i=1}^{n} w_i x_i = W$. The value of solution is $P(X) = \sum_{i=1}^{n} p_i x_i$

Now let $Y = (y_1, y_2, \dots, y_n)$ be any feasible solution. Since Y is feasible, $\sum_{i=1}^{n} w_i y_i \le W$

Hence $\sum_{i=1}^{n} (x_i - y_i) w_i \ge 0$. Let the value of the solution Y be $P(Y) = \sum_{i=1}^{n} p_i y_i \le W$

$$\text{Now } P(X) - P(Y) = \sum_{i=1}^{n} (x_i - y_i) p_i = \sum_{i=1}^{n} (x_i - y_i) w_i \frac{p_i}{w_i}$$

When $i<j$, $x_i = 1$ and so $(x_i - y_i)$ is positive or zero.
$i>j$, $x_i = 0$ and so $(x_i - y_i)$ is negative or zero. But since $p_i/w_i \le p_j/w_j$ in every case $(x_i - y_i)\, p_i/w_i \ge (x_i - y_i)\, p_j/w_j$
$i=j$, $p_i/w_i = p_j/w_j$

$$\text{Hence } P(X) - P(Y) \ge \frac{p_j}{w_j} \sum_{i=1}^{n} (x_i - y_i) w_i \ge 0$$

So we have proved that no feasible solution can have a value greater than P(X), so solution X is optimal.

3.2 Minimum Cost Spanning Trees

A spanning tree T = (V,E') of a connected, undirected graph G = (V,E) is a tree composed of all the vertices and some (or perhaps all) of the edges of *G*. A spanning tree of a connected graph G can also be defined as a maximal set of edges of G that contains no cycle, or as a minimal set of edges that connect all vertices.

A Minimum cost spanning tree is a subset T of edges of G such that all the vertices remain connected when only edges in T are used, and sum of the lengths of the edges in T is as small as possible. Hence it is then a spanning tree with weight less than or equal to the weight of every other spanning tree.

We know that the number of edges needed to connect an undirected graph with n vertices is n-1. If more that n-1 edges are present, then there will be a cycle. Then we can remove an edge which is a part of the cycle without disconnecting T. This will reduce the cost. There are two algorithms to find minimum spanning trees. They are Prim's algorithm and Kruskal's algorithm.

Prim's Algorithm

Prim's algorithm finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the

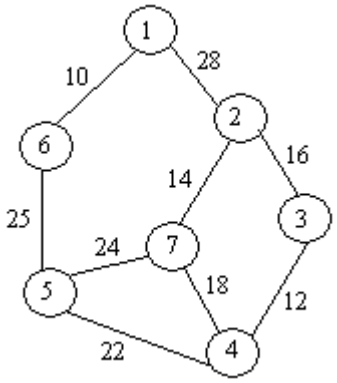
total weight of all the edges in the tree is minimized. The algorithm was discovered in 1930 by mathematician Vojtěch Jarník and later independently by computer scientist Robert C. Prim in 1957 and rediscovered by Edsger Dijkstra in 1959. Therefore it is sometimes called the DJP algorithm, the Jarník algorithm, or the Prim-Jarník algorithm.

Steps

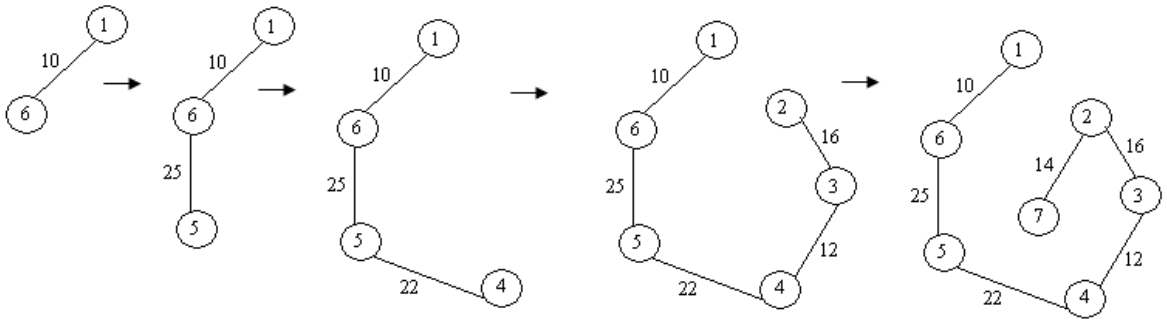
- Builds the tee edge by edge
- Next edge to be selected is one that result in a minimum increase in the sum of costs of the edges so far included
- Always verify that resultant is a tree

Example 3.2

Consider the connected graph given below



Minimum spanning tree using Prim's algorithm can be formed as given below.



Algorithm 3.2

```
float Prim (int e[][], float cost[][])
{
    int near[], j, k, l, I;
    float mincost = cost[k][L];      // (k, L) is the edge with minimum cost
    t[1][1] = k;
    t[1][2] = L;//L
    for (i=1; i<=n; i++)
       if (cost[i][L] < cost[i][k]) near[i] =L;
       else near[i] = k;
    near[k] = 0;
    near[L] = 0;
    for (i=2; i<=n; i++)
    {
```

```
            find j such that cost[j][near[j]] is minimum
            t[i][1] = j;
            t[i][2] = near[j];
            mincost = mincost + cost[j][near[j]];
            near[j] = 0;
            for (k=1; k<=n; k++)
                if (near[k] != 0 && cost[k][near[k] > cost[k][j])
                    near[k] = j;
    }
    return (mincost);
}
```

Time complexity of the above algorithm is $O(n^2)$.

Kruskal's Algorithm

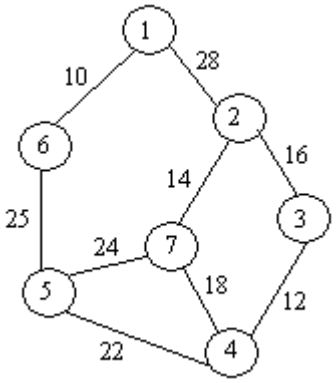
Kruskal's algorithm is another algorithm that finds a minimum spanning tree for a connected weighted graph. If the graph is not connected, then it finds a *minimum spanning forest* (a minimum spanning tree for each connected component).

Kruskal's Algorithm builds the MST in forest. Initially, each vertex is in its own tree in forest. Then, algorithm considers each edge in turn, order by increasing weight. If an edge (*u*, *v*) connects two different trees, then (*u*, *v*) is added to the set of edges of the MST, and two trees connected by an edge (*u*, *v*) are merged into a single tree on the other hand, if an edge (*u*, *v*) connects two vertices in the same tree, then edge (*u*, *v*) is discarded. The resultant may not be a tree in all stages. But can be completed into a tree at the end.

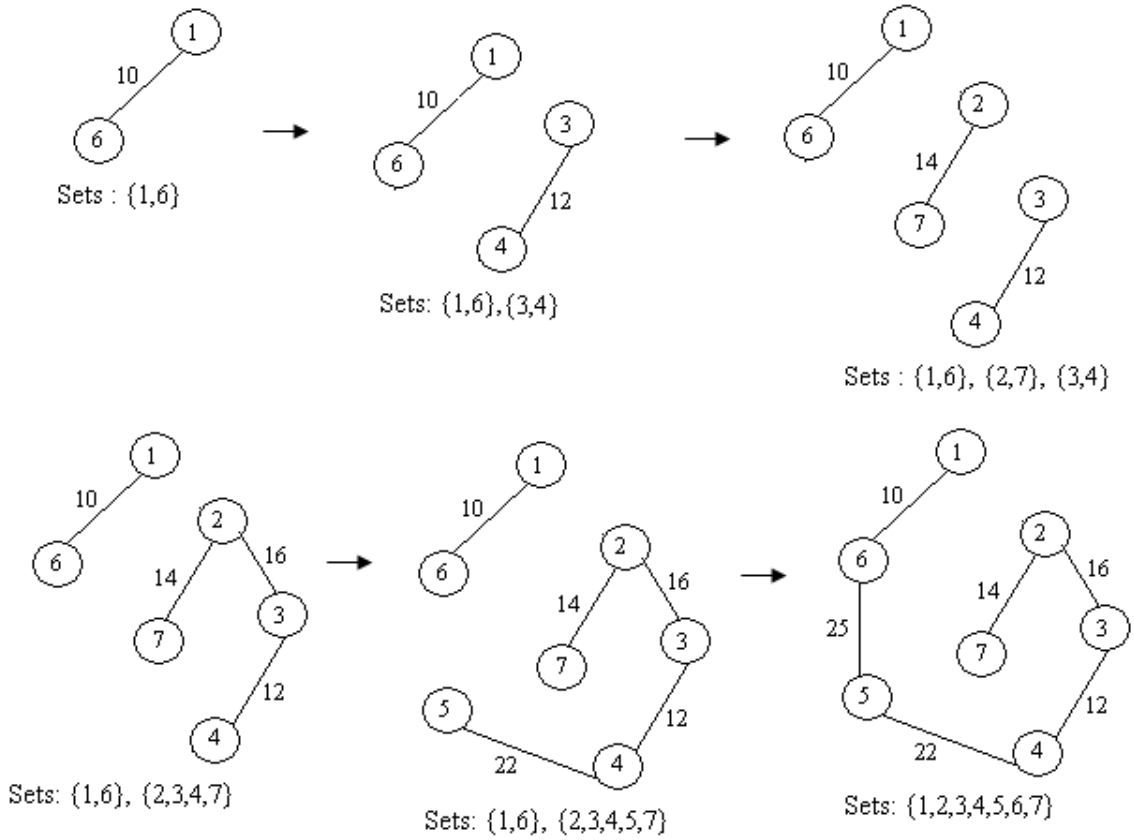
```
t = EMPTY;
while ((t has fewer than n-1 edges) && (E != EMPTY))
{
    choose an edge(v, w) from E of lowest cost;
    delete (v, w) from E;
    if (v, w) does not create a cycle in t
            add (v, w) to t;
    else
            discard (v, w);
}
```

To check whether there exist a cycle, place all vertices in the same connected component of *t* into a set. Then two vertices *v* and *w* are connected in *t* then they are in the same set.

Example 3.3
Consider the connected graph given below



Minimum spanning tree using Kruskal's algorithm can be formed as given below.



Algorithm 3.3

```
Float kruskal (int E[][], float cost[][], int n, int t[][2])
{
    int parent[w];
    consider heap out of edge cost;
    for (i=1; i<=n; i++)
            parent[i] = -1;          //Each vertex in different set
    i=0;
    mincost = 0;
    while((i<n-1) && (heap not empty))
    {
```

```
            Delete a minimum cost edge (u,v) from the heap and reheapify;
            j = Find(u);     k = Find(v);              // Find the set
            if (j != k)
            {
                    i++;
                    t[i][1] = u;
                    t[i][2] = v;
                    mincost += cost[u][v];
                    Union(j, k);
            }
            if (i != n-1)     printf("No spanning tree \n");
            else return(mincost);
    }
}
```

Time complexity of the above algorithm is O(n log n)

.

Theorem 3.2

Kruskal's algorithm will generate minimum cost spanning tree for every connected undirected graph G.

Proof

Let t represent spanning trees generated by Kruskal's algorithm and t' represent the minimum cost spanning tree for the given graph G. We have to prove that t and t' have same cost. Let E(t) and E(t') represents the edges in t and t' respectively.
If E(t) = E(t'), then t is a minimum cost spanning tree.

If E(t) ≠ E(t'), then let *q* be a minimum cost edge in *t* not in *t'*. i.e., q ∈ E(t) and $q \notin E(t')$. Inclusion of *q* into *t'* create a cycle in *t'*. say q, e_1, e_2 , … , e_k $1 \le i \le k$.

Let e_j be an edge in this cycle, not in E(t) i.e., $e_j \notin E(t)$. We know that e_j have cost greater than *q*. Because if e_j have cost lesser than *q*, kruskal's would have considered before *q*.
Hence all edges in E(t) of cost lesser than cost of *q* are also in E(t'). Now consider E(t') ∪{q}. Removal of any edge of cycle q, e_1, e_2 , … , e_k from create the tree is minimal cost. If we delete e_j, *t''* will have cost no more than *t'*. Hence *t''* is minimal cost. Doing this repeatedly will transform *t'* into *t* without any increase in cost. Hence is a minimum cost spanning tree.

3.3 Job Sequencing with Deadlines

Problem: Given a set of *n* jobs. Associated with each job *i* is a profit $p_i > 0$ and deadline $d_i \ge 0$. Profit is earned if job is completed before deadline. To complete a job, it has to be process one unit time. Only one machine is available to process the jobs. Feasible solution is a subset J of jobs such that each job in the subset can be completed by its deadline. The value of feasible solution is the sum of profits of jobs in J, $\sum_{i \in J} p_i$. An optimal solution is a feasible solution with maximum value.

Example 3.4

n = 4 $(p_1, p_2, p_3, p_4) = (100, 10, 15, 27)$ $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$

Jobs	Profit
(1, 2)	110
(1, 3)	115
(1, 4)	**127**
(2, 3)	25
(3, 4)	42

Arrange jobs in the decreasing order of profit. Then start takings jobs one by one.
Arranging the jobs in decreasing order we have (J_1, J_4, p_3, J_2).
Consider J_1, since deadline for J_1 is 2, include J_1 in the resultant set $\{J_1\}$
Now consider J_4. Since deadline J_4 is 1, we can include J_4 also in the resultant set by interchanging the order as $\{J_4 , J_1\}$. We cannot include J_3 as the deadline for J_3 is 1.
Hence the sequence is $J = \{J_4 , J_1\}$.

```
Greedy Jobs (int d[], set J,  int n)
{
    //Jobs are arranged in the decreasing order of profit.
    J = {1};          // J1 is the job with maximum profit
    for (int i; i<=n; i++)
        if (all jobs in J U{i} can be completed by their deadlines)  J = J U{i};
}
```

Algorithm 3.4

```
void JS (int d[], int j[], int n)
{
    // d[i] ≥1, n ≥1, p[1] ≥ p[2] ≥...≥ p[n]. We the algorithm terminates d[j[i]] ≤ d[j[i+1]]
    d[0] = j[0] = 0;
    k = 1;
    for ( int i=2; i<=n; i++)
    {
            r = k;
            while (d[j[r] > max(d[i], r))    r = r – 1;
            if (d[i] > r)
            {
                    for (m=k; m>=r+1; m--)        j[m+1] = j[m];
                    j[r+1] = i;
                    k = k+1;
            }
    }
    return;
}
```

Time complexity of the program is $\Theta(sn)$, where s is the number of terms in the result set.
Hence we can write the worse case complexity as $O(n^2)$.

Dynamic Programming

Dynamic programming is a method of solving problems exhibiting the properties of overlapping sub problems and optimal substructure that takes much less time than naive methods. The term was originally used in the 1940s by Richard Bellman to describe the process of solving problems where one needs to find the best decisions one after another.

It is used when the solution to a problem can be viewed as the result of a sequence of decisions. It avoid duplicate calculation in many cases by keeping a table of known results and fills up as sub instances are solved.

It follows a bottom-up technique by which it start with smaller and hence simplest sub instances. Combine their solutions to get answer to sub instances of bigger size until we arrive at solution for original instance.

Dynamic programming differs from Greedy method because Greedy method makes only one decision sequence but dynamic programming makes more than one decision.

Principle of Optimality
An optimal sequence of decisions has the property that whatever the initial state and decisions are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision.

All pairs Shortest Path

Let G=(V,E) be a directed graph with n vertices.The cost of the vertices are represented by an adjacency matrix such that $1 \le i \le n$ and $1 \le j \le n$

cost(i, i) = 0,
cost(i, j) is the length of the edge <i, j> if <i, j> $\in$ E and
cost(i, j) = ∞ if <i, j> $\notin$ E.

The graph allows edges with negative cost value, but negative valued cycles are not allowed. All pairs shortest path problem is to find a matrix A such that A[i][j] is the length of the shortest path from i to j.

Consider a shortest path from i to j, $i \neq j$. The path originates at i goes through possibly many vertices and terminates at j. Assume that there is no cycles. If there is a cycle we can remove them without increase in the cost. Because there is no cycle with negative cost.
Initially we set A[i][j] = c[i][j].
Algorithm makes n passes over A. Let A_0, A_1, .. A_n represent the matrix on each pass.
Let $A^{k-1}[i,j]$ represent the smallest path from i to j passing through no intermediate vertex greater than k-1. This will be the result after k-1 iterations. Hence k^{th} iteration explores whether k lies on optimal path. A shortest path from I to j passes through no vertex greater than k, either it goes through k or does not.
If it does , $A^k[i,j] = A^{k-1}[i,k] + A^{k-1}[k,j]$
If it does not, then no intermediate vertex has index greater than k-1. Then $A^k[i,j] = A^{k-1}[i,j]$

Combining these two conditions we get
$A^k[i,j] = \min\{ A^{k-1}[i,j], A^{k-1}[i,k] + A^{k-1}[k,j]\}$, k>=1 ($A^0[i,j]$= cost(i, j))

Algorithm 4.2

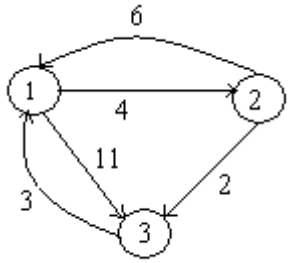
```
Void AllPaths ( float cost[][], float A[][], int n)
{
    for (int i=1; i<=n; i++)
        for (int j=1; j<=n; j++)
            A[i][j] = cost[i][j];

    for (int k=1; k<=n; k++)
        for (i=1; i<=n; i++)
            for (j=1; j<=n; j++)
                    A[i][j] = min{A[i][j], A[i][k] + A[k][j]);
}
```

Time complexity of the algorithm is $O(n^3)$.

Example 4.2

Consider the directed graph given below.



Cost adjacency matrix for the graph is as given below

$$\begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$$

Copy the cost values to the matrix A. So we have A_0 as

A_0	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

Matrix A after each iteration is as given below.

A_1	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

A_2	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

A_3	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

The Traveling Salesman problem

Problem: A salesman spends his time in visiting cities cyclically. In his tour he visits each city just once, and finishes up where he started. In which order should he visit to minimize the distance traveled.

The problem an be represented using a graph. Let G=(V,E) be a directed graph with cost c_{ij}, c_{ij} >0 for all <i j> $\in$ E and $c_{ij} = \infty$ for all <j,t> $\notin$ E. |V| = n and n>1. We know that tour of a graph includes all vertices in V and cost of tour is the sum of the cost of all edges on the tour. Hence the traveling salesman problem is to minimize the cost.

Application

The traveling salesman problem can be correlated to many problems that we find in the day to day life. For example, consider a production environment with many commodities manufactured by same set of machines. Manufacturing occur in cycles. In each production cycle n different commodities are produced. When machine changes from product *i* to product *j*, a cost C_{ij} is incurred. Since products are manufactured cyclically, for the change from last commodity to the first a cost is incurred. The problem is to find the optimal sequence to manufacture the products so that the production cost is minimum.

We know that the tour of the simple graph starts and ends at vertex1. Every tour consist of an edge <i, k> for some k$\in$ V-{1} and a path from k to 1. Path from k to 1 goes through each vertex in V- {1,k} exactly once. If tour is optimal, path from k to 1 muat be shortest k to 1 path going through all vertices in V-{1,k}. Hence the principle of optimality holds.

Let g(i, S) be length of shortest path starting at i, going through all vertices in S ending at 1. Function g(1, V-{1}) is the optimal salesman tour. From the principle of optimality

$$g(1, V-\{1\}) = \min_{2\le k\le n} \{C_{1k} + g(k, V-\{1,k\})\} \quad \text{——} \quad (1)$$

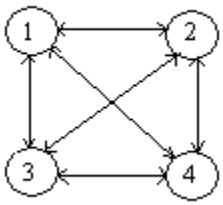
$$\text{for } i \notin S \quad g(1, S) = \min_{j\in S} \{C_{ij} + g(j, S-\{j\})\} \quad \text{————} \quad (2)$$

$$g(i, \varphi) = C_{i1}, \ 1\le i\le n$$

Use eq.(1) to get g(i, S) for |S| =1. Then find g(i, S) with |S|=2 and so on.

Example 4.3

Consider the directed graph with the cost adjacency matrix given below.



$$C_{ij} = \begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

$g(2, \varphi) = C_{21} = 5$
$g(3, \varphi) = C_{31} = 6$
$g(4, \varphi) = C_{41} = 8$

$g(2, \{3\}) = C_{23} + g(3, \varphi) = 15$
$g(2, \{4\}) = C_{24} + g(4, \varphi) = 18$
$g(3, \{2\}) = C_{32} + g(2, \varphi) = 18$
$g(3, \{4\}) = C_{34} + g(4, \varphi) = 20$
$g(4, \{2\}) = C_{42} + g(2, \varphi) = 13$

$g(4, \{3\}) = C_{43} + g(3, \varphi) = 15$

$g(2, \{3,4\}) = \min\{C_{23} + g(3, \{4\}) , C_{24} + g(4, \{3\}) \}= 25$
$g(3, \{2,4\}) = \min\{C_{32} + g(2, \{4\}) , C_{34} + g(4, \{2\}) \}= 25$
$g(4, \{2,3\}) = \min\{C_{42} + g(2, \{3\}) , C_{43} + g(3, \{2\}) \}= 23$

$g(1, \{2,3,4\}) = \min\{C_{12} + g(2, \{3,4\}) , C_{13} + g(3, \{2,4\}) , C_{14} + g(4, \{2,3\}) \}$
$= \min\{35, 40, 43\} = 35$

Let J(i, S) represent the value of j which is the value of g(i, S) that minimizes the right hand side of equ(2). Then J(1, {2,3,4}) =2, which infers that the tour starts from 1 goes to 2. Since J(2,{3,4}) = 4, we know that from 2 it goes to 4. J(4, {3}) = 3. Hence the next vertex is 3. Then we have the optimal sequence as **1,2,4,3,1**.