**Chapter -3 Greedy Method**

* **General Method**
* **Knapsack Problem**
* **Minimum Cost Spanning Tree –Kruskal and primal Algo**
* **Single Source Shorted Path**
* **Job Sequencing with deadlines**
* **Optimal Storage on tapes**

**General Method :**

* An algorithm which always takes the best immediate, or local, solution while finding an answer. Greedy algorithms will always find the overall, or globally, *[optimal solution](file:///C:\\class\\optimalsoltn.html)* for some *[optimization problems](file:///C:\\class\\optimization.html)*, but may find less-than-optimal solutions for some instances of other problems.

**Example of Greedy Method**

* [***Prim's algorithm***](file:///C:\class\primsalgrthm.html) ***and*** [***Kruskal's algorithm***](file:///C:\class\kruskalsalgo.html) ***are greedy algorithms which find the globally optimal solution, a [minimum spanning tree](file:///C:\\class\\minspantree.html). In contrast, any known greedy algorithm to find an [Euler cycle](file:///C:\\class\\eulercycle.html) might not find the shortest path, that is, a solution to the [traveling salesman](file:///C:\\class\\travelsales.html) problem.***
* [***Dijkstra's algorithm***](file:///C:\class\dijkstraalgo.html) ***for finding [shortest paths](file:///C:\\class\\shortestpath.html) is another example of a greedy algorithm which finds an optimal solution.***

**Features**

* Start with a solution to a small sub problem
* Build up to a solution to the whole problem
* Make choices that look good in the short term
* Disadvantage: Greedy algorithms don’t always work ( Short term solutions can be disastrous in the long term). Hard to prove correct
* Advantage: Greedy algorithm work fast when they work. Simple algorithm, easy to implement

**Greedy Algorithm**

Procedure GREEDY(A,n)

// A(1:n) contains the n inputs//

solution ← φ //initialize the solution to empty//

for i ← 1 to n do

x ← SELECT(A)

if FEASIBLE(solution,x)

then solution ← UNION(solution,x)

end if

repeat

return(solution)

end GREEDY

**Knapsack Problem**

Greedy method is best suited to solve more complex problems such as a knapsack problem. In a knapsack problem there is a knapsack or a container of capacity M n items where, each item I is of weight wi and is associated with a profit pi. The problem of knapsack is to fill the available items into the knapsack so that the knapsack gets filled up and yields a maximum profit. If a fraction xi of object i is placed into the knapsack, then a profit pi \*xi is earned. The constrain is that all chosen objects should sum up to M.

OR

* **Problem definition**
  + Given n objects and a knapsack where object i has a weight wi and the knapsack has a capacity m
  + If a fraction xi of object i placed into knapsack, a profit pixi is earned

The objective is to obtain a filling of knapsack maximizing the total profit

* **Problem formulation (Formula 4.1-4.3)**



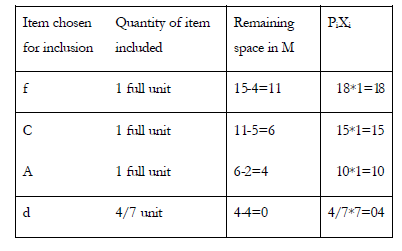
* **A *feasible solution* is any set satisfying (4.2) and (4.3)**
* **An *optimal solution* is a feasible solution for which (4.1) is maximized**
* Greedy selection policy: three natural possibilities
* Policy 1: Choose the lightest remaining item, and take as much of it as can fit.
* Policy 2: Choose the most profitable remaining item, and take as much of it as can fit.
* Policy 3: Choose the item with the highest price per unit weight (P[i]/W[i]), and take as much of it as can fit. =🡺Policy 3 always gives an optimal solution.

**Illustration**

Consider a knapsack problem of finding the optimal solution where, M=15, (p1,p2,p3…p7) = (10, 5, 15, 7, 6, 18, 3) and (w1,w2, …., w7) = (2, 3, 5, 7, 1, 4, 1).In order to find the solution, one can follow three different strategies.

**Strategy 1**: non-increasing profit values(Largest Profit)

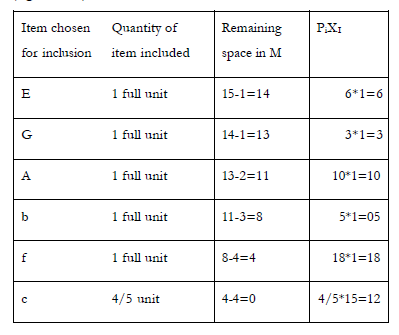
Let (a,b,c,d,e,f,g) represent the items with profit (10,5,15,7,6,18,3) then the sequence of objects with non increasing profit is (f,c,a,d,e,b,g).



Profit= 47 units The solution set is (1, 0, 1, 4/7, 0, 1, 0).

**Strategy 2**: non-decreasing weights(Smallest Wight)

The sequence of objects with non-decreasing weights is (e,g,a,b,f,c,d).

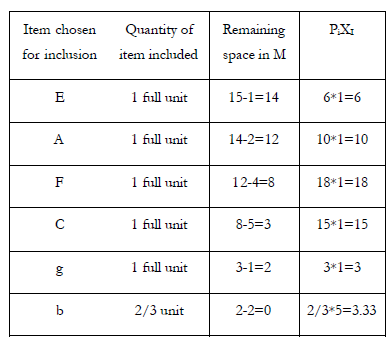


**Profit= 54 units The solution set is (1,1,4/5,0,1,1,1).**

**Strategy 3**: maximum profit per unit of capacity used (This means that the objects are considered in decreasing order of the ratio Pi/wI)

a: P1/w1 =10/2 = 5 b: P2/w2 =5/3=1.66 c: P3/w3 =15/5 = 3 d: P4/w4 =7/7=1 e: P5/w5 =6/1=6 f: P6/w6 =18/4 = 4.5 g: P7/w7 =3/1=3

Hence, the sequence is (e, a, f, c, g, b, d)



**Profit= 55.33 units The solution set is (1,2/3,1,0,1,1,1).**

**Example2**. n = 3, M = 20, (p1, p2, p3) = (25, 24, 15) (w1, w2, w3) = (18, 15, 10)

Sol: p1/w1 = 25/18 = 1.32

p2/w2 = 24/15 = 1.6

p3/w3 = 15/10 = 1.5

Optimal solution: x1 = 0, x2 = 1, x3 = 1/2

total profit = 24 + 7.5 = 31.5

**Algorithm GREEDY\_KNAPSACK (P,W,M,X,n)**

//P(1:n) and W(1:n) contain the profit and weights respectively of the n objects ordered so that P(i)/W(i) >=P(i+1)/W(i+1).M is the knapsack size and X(1:n) is the solution vector

Real P(1:n),W(1:n),X(1:n) ,M, cu;

Integer I,n;

X🡨 0 //initialize solution to Zero

Cu🡨M // cu is remaining knapsack capacity

for i🡨 1 to n do

if(W(i) >cu ) then exit endif

X(i)🡨1

Cu🡨 cu-W(i)

Repeat

If (i<=n ) then X(i) 🡨 cu/W(i) endif

End

**Time complexity**

* + - Sorting: O(n log n) using fast sorting algorithm like merge sort
    - GreedyKnapsack: O(n)
    - So, total time is O(n log n)

**Minimum Cost Spanning Tree –Kruskal and Prim’s Algo**

**Tree:**

– A tree is a graph with the followingproperties:

• The graph is connected (can go from anywhere to anywhere)

• There are no cycle

**Spanning Tree**

• A spanning tree is a tree that spans all the nodes Thus, if there are n nodes in the network, a tree spanning this network will have n-1 arcs that go through all the nodes.

**Minimum Spanning tree**

• It is the shortest spanning tree (length of a tree is equal to the sum of the length of the arcs on the tree).

• Very important

– Practice (eg. communication)

– Theory (eg. basis)

– Algorithms (as a sub problem)

Or

A **minimum spanning tree** (MST) or **minimum weight spanning tree** is then a spanning tree with weight less than or equal to the weight of every other spanning tree.

Or

* **Definition Let *G=(V, E*) be at undirected connected graph. A subgraph *t=(V, E’)* of *G* is a *spanning tree* of *G* iff t is a tree.**

**Example**

**Algorithm for a Spanning Tree**

• Two basic algorithms exists – Kruskal (by arc) – Prim (by sub-tree)

• Both are greedy

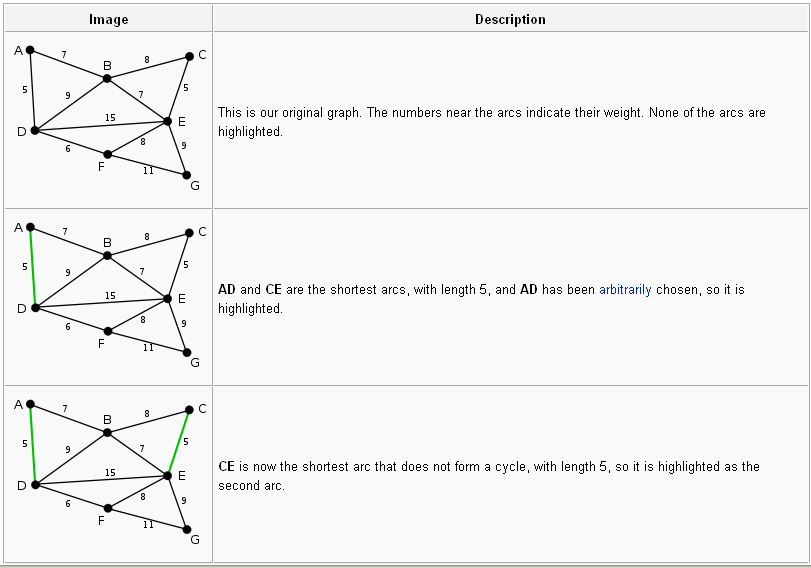
• May have different complexity (efficiency) depending on the topology (eg. density) of the network

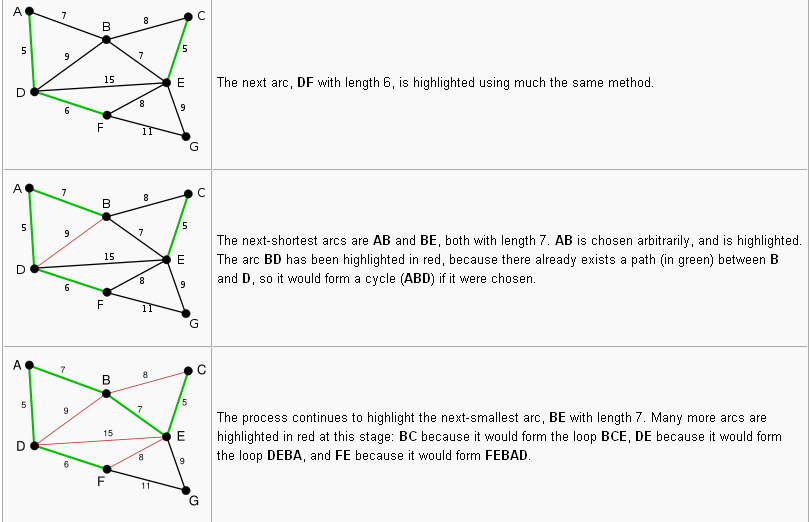
**Kruskal Algorithm**

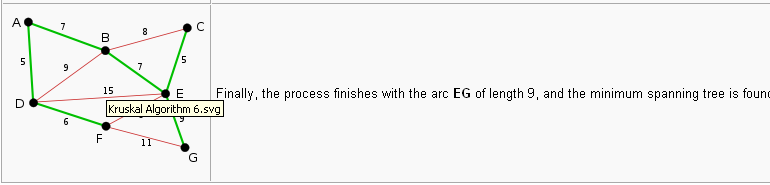
• Append new arcs to the tree in increasing order of the arc length (making sure cycles are not created)

Kruskal's algorithm is an [algorithm](http://en.wikipedia.org/wiki/Algorithm) in [graph theory](http://en.wikipedia.org/wiki/Graph_theory) that finds a [minimum spanning tree](http://en.wikipedia.org/wiki/Minimum_spanning_tree) for a connected weighted graph. This means it finds a subset of the [edges](http://en.wikipedia.org/wiki/Edge_(graph_theory)) that forms a tree that includes every [vertex](http://en.wikipedia.org/wiki/Vertex_(graph_theory)), where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a minimum spanning forest (a minimum spanning tree for each [connected component](http://en.wikipedia.org/wiki/Connected_component_(graph_theory))). Kruskal's algorithm is an example of a [greedy algorithm](http://en.wikipedia.org/wiki/Greedy_algorithm).

**Example**







**Algorithm :**

KRUSKAL ( E,Cost,n,T,mincost)

//E is the set of edges in G.

//G has n vertices.

//Cost(U,v) is the cost of edge (u,v).

//T is the set of edes in the minimum spanning tree and mincost is its cost

1. Real min cost, cost(1:n,1:n)
2. Integer PARENT(1:n) ,T(1:n-1,2) ,n construct a heap out of the edge costs using HEAPIFY
3. Parent🡨 1 // each vertex is in a different set
4. I🡨mincost 🡨 0
5. While i🡨n-1 and heap not empty do
6. Delete a minimum cost edge (u,v) from the heap and reheapify using ADJUST
7. J🡨 FIND (u) ;K🡨 FIND(V)
8. If j != k then i🡨 i+1
9. T(I,1) 🡨 u ;T(I,2) 🡨 v
10. Mincost 🡨mincost +cost (u,v)
11. Endif
12. Repeat
13. If i< > n-2 then print (“no spanning tree) end if
14. Return
15. End Kruskal

**How to implement -**

**Two functions should be considered**

* + - **Determining an edge with minimum cost**
    - **Deleting this edge**

**Analysis of Algorithm**

Where *E* is the number of edges in the graph and *V* is the number of vertices, Kruskal's algorithm can be shown to run in [*O*](http://en.wikipedia.org/wiki/Big-O_notation)(*E* [log](http://en.wikipedia.org/wiki/Binary_logarithm) *E*) time, or equivalently, *O*(*E* log *V*) time, all with simple data structures. These running times are equivalent because:

* *E* is at most *V*2 and log*V*2 = 2log*V* is *O*(log *V*).
* If we ignore isolated vertices, which will each be their own component of the minimum spanning forest, *V* ≤ *E*+1, so log *V* is *O*(log *E*).

We can achieve this bound as follows: first sort the edges by weight using a [comparison sort](http://en.wikipedia.org/wiki/Comparison_sort) in *O*(*E* log *E*) time; this allows the step "remove an edge with minimum weight from *S*" to operate in constant time. Next, we use a [disjoint-set data structure](http://en.wikipedia.org/wiki/Disjoint-set_data_structure) to keep track of which vertices are in which components. We need to perform O(*E*) operations, two 'find' operations and possibly one union for each edge. Even a simple disjoint-set data structure such as disjoint-set forests with union by rank can perform O(*E*) operations in *O*(*E* log *V*) time. Thus the total time is *O*(*E* log *E*) = *O*(*E* log *V*).

**Or**

Edge set E.

Operations are:

* + Is E empty?
  + Select and remove a least-cost edge.

Use a min heap of edges.

* + Initialize. O(e) time.
  + Remove and return least-cost edge. O(log e) time.

Set of selected edges T.

Operations are:

* + Does T have n - 1 edges?
  + Does the addition of an edge (u, v) to T result in a cycle?

Add an edge to T.

Use an array linear list for the edges of T.

* + Does T have n - 1 edges?
    - Check size of linear list. O(1) time.
  + Does the addition of an edge (u, v) to T result in a cycle?
    - Not easy.
  + Add an edge to T.
    - Add at right end of linear list. O(1) time.

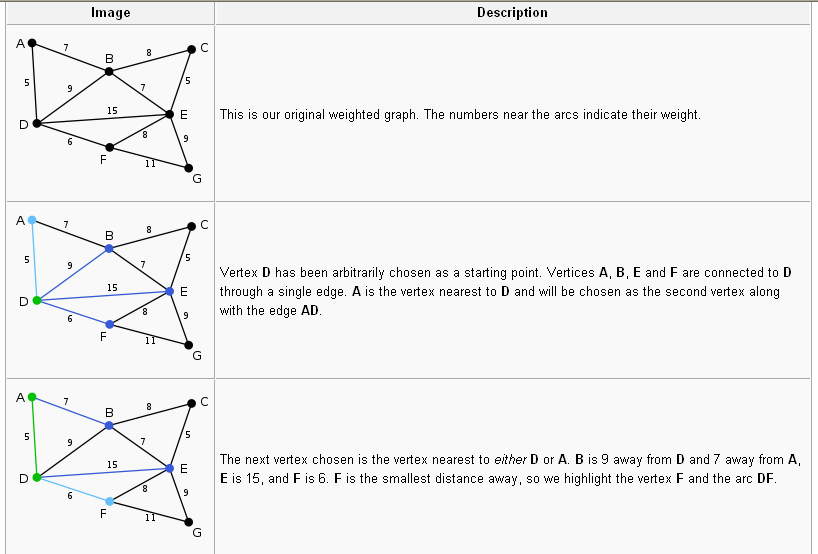
Just use an array rather than ArrayLinearList

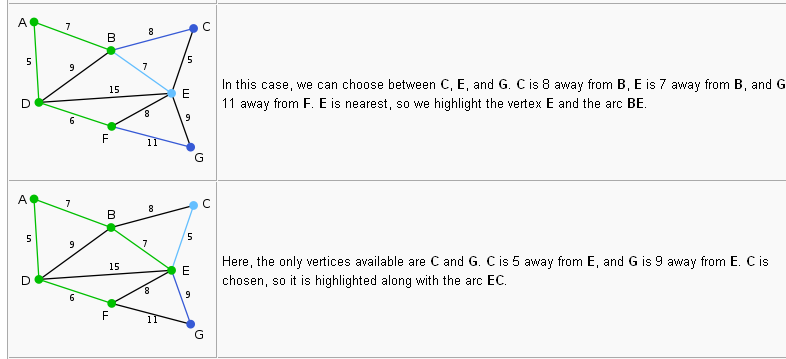
* Use FastUnionFind.
* Initialize.
  + O(n) time.
* At most 2e finds and n-1 unions.
  + Very close to O(n + e).
* Min heap operations to get edges in increasing order of cost take O(e log e).
* Overall complexity of Kruskal’s method is O(n + e log e).

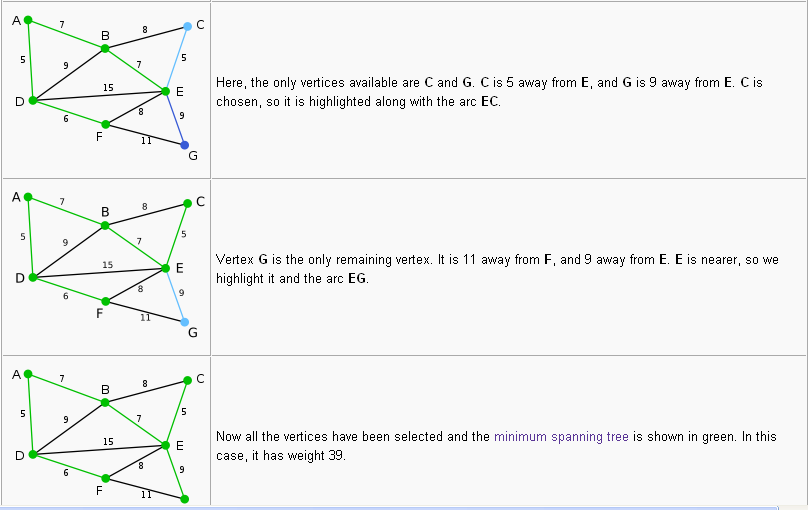
**Prim’s Algorithm**

• Similar, except that we always have a sub-tree as a partial solution: the new arc we add connects a node in the existing sub-tree to a node not yet in the sub-tree.

Example







**Algorithm**

**PRIME(E,COST,nT,mincost)**

**//**E is the set of edges in G

//COST (n,n) is the cost adjacency matrix of an n vertex graph such that COST(I,j) is either a positive real number + infinity. If no edge exists. A minimum spanning tree is computed and stored as set of edges in the array T(1:n-1,2). T(1,1)T(I,2) is an edge in the min-cost spanning tree .The final cost is assigned to mincost

1. real COST( n,n) ,mincost ;
2. integer NEAR(n) ,n,I,j,k,l ,T(1:n-1,2);
3. (K,l) 🡨 edge with minimum cost
4. mincost = cost(k,l);
5. T(1,1),T(1,2) 🡨(k,l)
6. for I 🡨1 to n do // Initialize near.
7. If COST(i ,l)< COST (i, k) then NEAR (i) 🡨 l
8. Else NEAR(i) 🡨 k endif
9. Repeat
10. NEAR (k) 🡨 NEAR (l) 🡨 0
11. for I 🡨 2 to n-1 do //find n-2 additional edges for T.
12. // Let j be an index such that NEAR (J)!= 0 and COST (j ,NEAR(j)) is minimum
13. (T(i,1) ,T(i,2)) 🡨 (J ,NEAR (j))
14. mincost 🡨mincost +COST (j,NEAR(j))
15. NEAR (j) 🡨 0
16. for K 🡨1 to n do //update NEAR
17. if NEAR(K) != 0 and COST (K ,NEAR (k)) > COST(K,j) then
18. NEAR(K) 🡨 j
19. End if
20. Repeat
21. Repeat
22. If mincost >= infinity then print (“no spanning tree )
23. End PRIM

**Time complexity**

|  |  |
| --- | --- |
| **Minimum edge weight data structure** | **Time complexity (total)** |
| [adjacency matrix](http://en.wikipedia.org/wiki/Adjacency_matrix), searching | O(VE) |

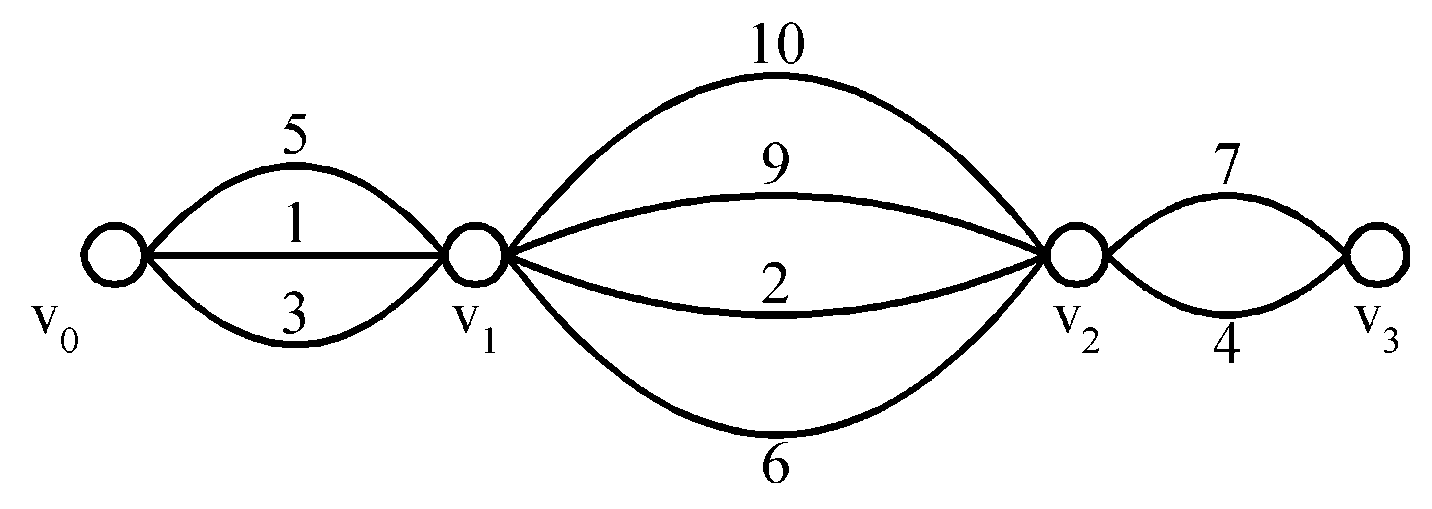
* **single-source shortest path (Dijstra’s Algorithm)**

In [graph theory](http://en.wikipedia.org/wiki/Graph_theory), the **shortest path problem** is the problem of finding a [path](http://en.wikipedia.org/wiki/Path_(graph_theory)) between two [vertices](http://en.wikipedia.org/wiki/Vertex_(graph_theory)) (or nodes) such that the sum of the [weights](http://en.wikipedia.org/wiki/Glossary_of_graph_theory#Weighted_graphs_and_networks) of its constituent edges is minimized

* Directed weighted graph.
* Path length is sum of weights of edges on path.
* The vertex at which the path begins is the source vertex.
* The vertex at which the path ends is the destination vertex.

**Example**

Finding the quickest way to get from one location to another on a road map; in this case, the vertices represent locations and the edges represent segments of road and are weighted by the time needed to travel that segment.



* Problem: Find a shortest path from v0 to v3.
* The greedy method can solve this problem.
* The shortest path: 1 + 2 + 4 = 7.

The problem is also sometimes called the **single-pair shortest path problem**, to distinguish it from the following generalizations:

* The **single-source shortest path problem**, in which we have to find shortest paths from a source vertex *v* to all other vertices in the graph.
* The **single-destination shortest path problem**, in which we have to find shortest paths from all vertices in the graph to a single destination vertex *v*. This can be reduced to the single-source shortest path problem by reversing the edges in the graph.
* The **all-pairs shortest path problem**, in which we have to find shortest paths between every pair of vertices *v*, *v'* in the graph.

**Single Source Shortest Path :**

* **Design of greedy algorithm**

Building the shortest paths one by one, in non decreasing order of path lengths

e.g., in Figure 4.15

1🡪4: 10

1🡪4🡪5: 25

…

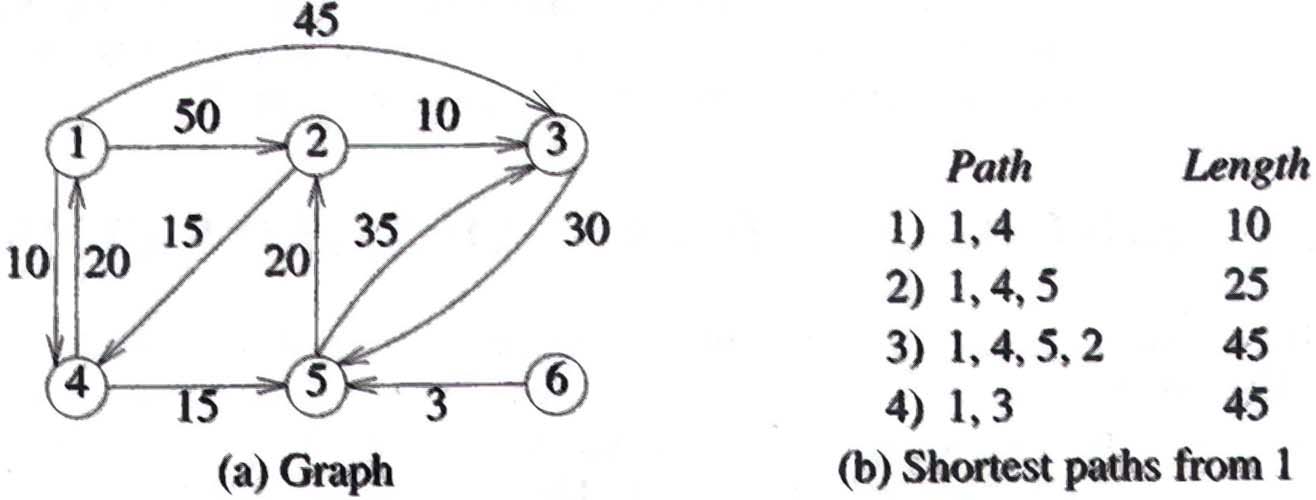
We need to determine 1) the next vertex to which a shortest path must be generated and 2) a shortest path to this vertex

**Three observations**

If the next shortest path is to vertex *u*, then the path begins at *v0,* ends at *u*, and goes through only those vertices that are in *S*.

The destination of the next path generated must be that of vertex *u* which has the minimum distance, *dist(u),* among all vertices not in *S*.

Having selected a vertex *u* as in observation 2 and generated the shortest *v0* to *u* path, vertex *u* becomes a member of *S*.

****

**Algorithm :Greedy algorithm ( Dijkstra’s algorithm)**

ShortestPaths(v, cost, dist, n)

//DIST(j) ,1<=j<=n is set to the length of the shortest path from vertex v to vertex j in a diagraph G with n vertices. DIST(v) is set to zero. G is represented by its cost adjacency matrix ,COST(n,n)

Boolean S(1:n); real COST(1:n,1:n) DIST(1:n)

Integer u,v, n,num ,I,w

For I 🡨 to n do //initialize set S to empty

S(i)🡨0 ; DIST(i) 🡨 COST(V,i)

Repeat

S(V) 🡨1 ;DIST(v) 🡨 0 //put vertex v in set S

For num 🡨 2 to n-1 do //determine n-1 paths from vertex v //

Choose u such that DIST (u) =min {DIST(w)}

S(w)=0

S(u)🡨1 //put vertex u in set s

For all W with S(w) =0 do

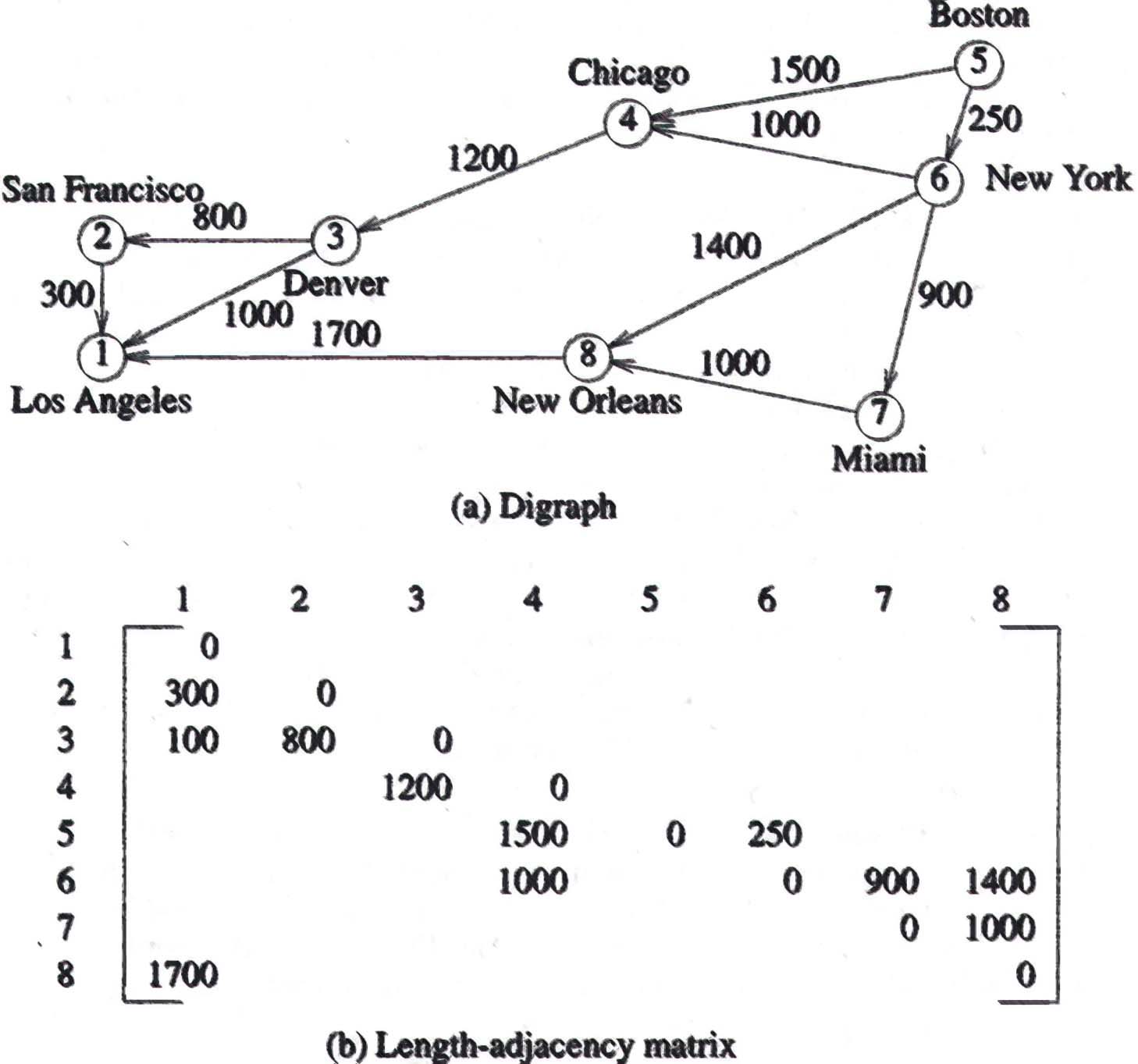
**DIST(w) 🡨 min(DIST(w) ,DIST( u) +COST(u,w))**

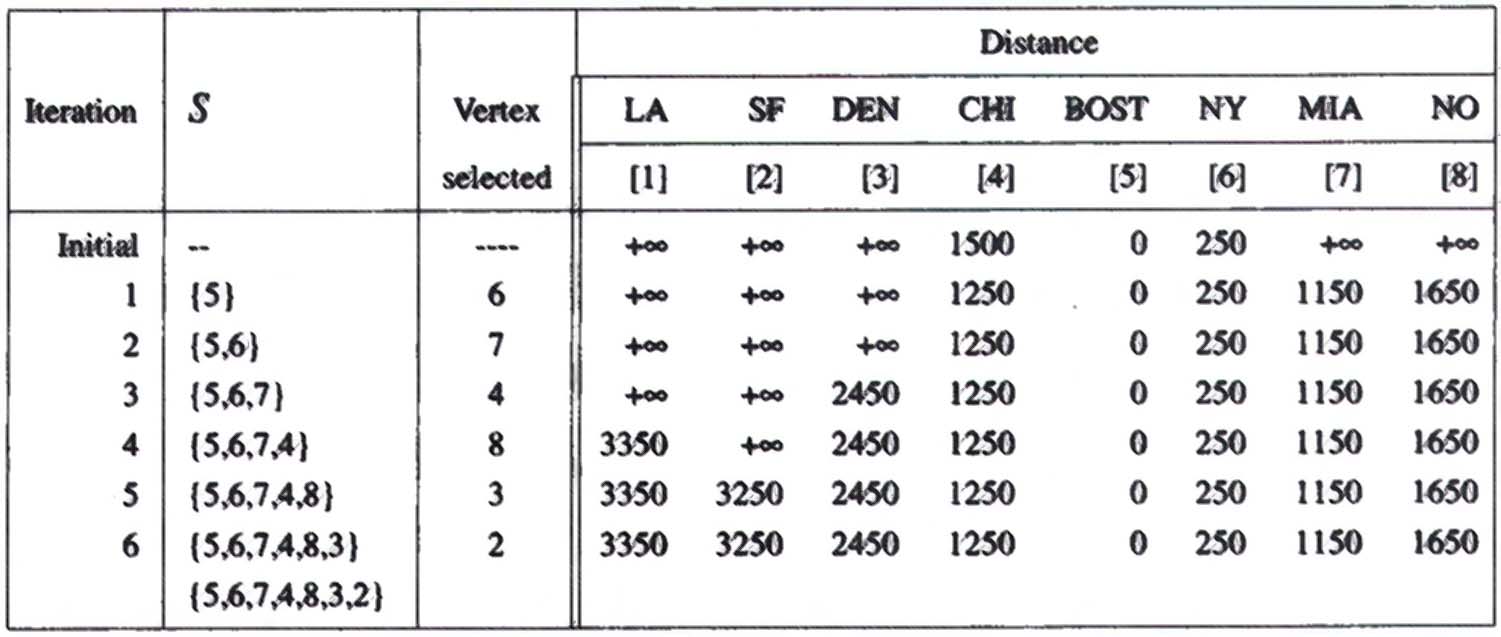
Repeat

Repeat

End SHORTEST-PATHS.

**Time Complexity :- O(n2)**

****

****