Ch 4 Greedy method

4.1 Basic concept

Greedy concept

- When making decisions, you always choose the best one for yourself
- → The decision should be a feasible solution
  - The solution should fit constraints (or say rules)
- → The target problem has an objective function
- → If the decision can achieve the best result → optimal solution

Two types of greedy

- Subset paradigm
  - Make decision in every stage
  - May have suboptimal solution

- Ordering paradigm
  - Decide a sequence of decisions
  - To observe if the optimal result can be achieved

Ex 4.1 change making

- 67 cents in change
- 67 → 2 quarters (25 cents)
  → 1 dime (10 cents)
  → 1 nickel (5 cents)
  → 2 pennies (1 cent)

Ex 4.2 machine schedule

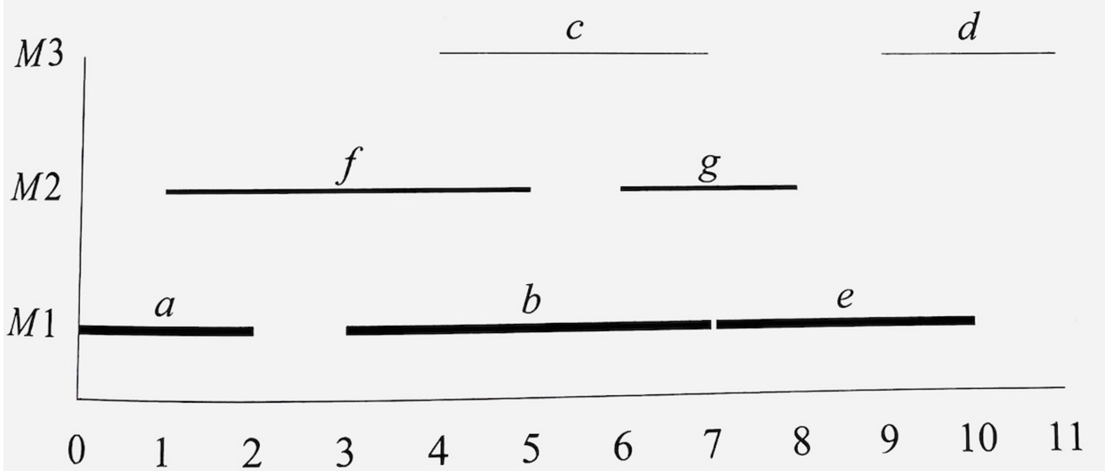
- There *n* tasks
- Each task has a start time s_i and finish time f_i
- The condition of feasible is no overlap task
  - i.e., the last task is not finished and another task is launched
- The objective function is to use as less machine to schedule all tasks

The policy

- Sort each task by their start time
- Check if a task can be added to an existing machine
    - If yes, add the task
    - If no, use a new machine

task	*a*	*b*	*c*	*d*	*e*	*f*	*g*
start	0	3	4	9	7	1	6
finish	2	7	7	11	10	5	8

(a) Seven tasks



4.2 container loading

Cargo problem

- Give n cargo
- Each container has different weights

  → w_i is the weight of container i
- The capacity of the ship is C

- x_i = 1 represents the container is selected
- x_i = 0 represents the container is not selected
- $\sum_{i=1}^{n} x_i w_i \leq C$ → The target is to maximize $\sum_{i=1}^{n} x_i$
  - i.e., to carry more container

policy

- Sort w_i in increasing order

- Select containers by the above sequence

  - When selecting a container → check if feasible
  - If feasible → select the next one and check again

```
Algorithm ContainerLoading(c, capacity, numberOfContainers, x)
// Greedy algorithm for container loading.
// Set x[i] = 1 iff container c[i], i ≥ 1 is loaded.
{
    // sort into increasing order of weight
    Sort(c, numberOfContainers);

    n := numberOfContainers;

    // initialize x
    for i := 1 to n do
        x[i] := 0;

    // select containers in order of weight
    i := 1;
    while (i ≤ n && c[i].weight ≤ capacity)
    {
        // enough capacity for container c[i].id
        x[c[i].id] := 1;
        capacity− = c[i].weight; // remaining capacity
        i + +;
    }
}
```

4.3 knapsack problem

knapsack problem

- There are n objects and one bag
- The weight of object i is w_i
- The capacity of the bag is m
- Objects can be divided, and part of the object x_i can be put in the bag
  - where $0 \leq x_i \leq 1$

- When part of the object i in the bag, we can have reward $p_i x_i$

- The object is to maximize the total reward

The linear equation

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

$$\text{subject to} \sum_{1 \le i \le n} w_i x_i \le m$$

$$\text{and } 0 \le x_i \le 1, \quad 1 \le i \le n$$

The policy

- Select objects by the portion of objects' weights
- Select objects with larger profits first
- Select objects with smaller weights first
- Select objects with maximum unit profits (more contributions)

Example 4.1 Consider the following instance of the knapsack problem: $n = 3, m = 20, (p_1, p_2, p_3) = (25, 24, 15)$, and $(w_1, w_2, w_3) = (18, 15, 10)$. Four feasible solutions are:

	(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum p_i x_i$
1.	(1/2, 1/3, 1/4)	16.5	24.25
2.	(1, 2/15, 0)	20	28.2
3.	(0, 2/3, 1)	20	31
4.	(0, 1, 1/2)	20	31.5

Of these four feasible solutions, solution 4 yields the maximum profit. As we shall soon see, this solution is optimal for the given problem instance. □

Algo 4.3

```
Algorithm GreedyKnapsack(m, n)
// p[1 : n] and w[1 : n] contain the profits and weights respectively
// of the n objects ordered such that p[i]/w[i] ≥ p[i + 1]/w[i + 1].
// m is the knapsack size and x[1 : n] is the solution vector.
{
    for i := 1 to n do x[i] := 0.0; // Initialize x.
    U := m;
    for i := 1 to n do
    {
        if (w[i] > U) then break;
        x[i] := 1.0; U := U - w[i];
    }
    if (i ≤ n) then x[i] := U/w[i];
}
```

4.4 tree vertex splitting

Scenario

- In real world, transmissions are prone to have losses
- When the loss exceeds a predefined threshold → need a booster

- The question is how / where to place boosters

Model

- Let T=(V, E, w) is a directed binary tree
  - w(i, j) is the weight between edge (i, j)
- In the tree,
  - some nodes are source nodes → in-degree = 0
  - some nodes are sink nodes → out-degree = 0
- For a path P (in the tree), d(P) is the sum of the weights on P
  - d(T) is $\max_{\forall p \in T} d(P)$, i.e., the max d(P) in the tree T will be d(T)

- A tree node u can be split, the u can become two nodes
  - u^o: the new source node / u^i: the new sink node
  - → the node u is the booster

- Target → split the T into multiple trees (i.e., forest)
  - Each tree T/X will satisfy $d(T/X) \le \delta$ (The δ is the threshold for allowable loss)

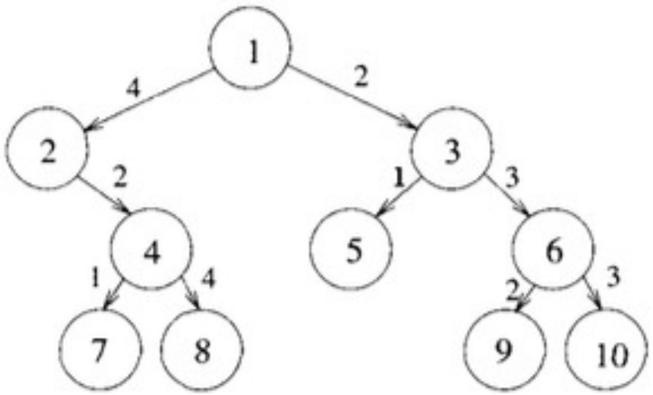
- There are |V| nodes → there are possible $2^{|V|}$ possibilities for splitting

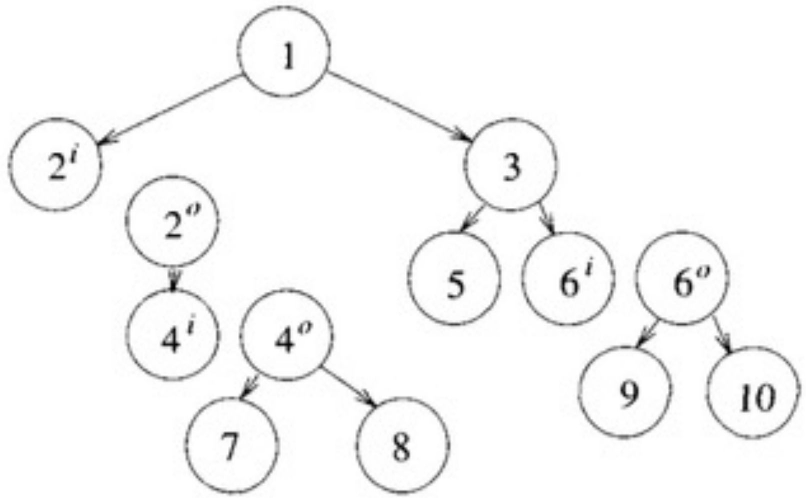
The greedy policy

- For a node u, compute the maximum delay from u to those descendants in u's subtree, say d(u)

$$d(u) = \max_{x \in c(u)} \{d(x) + w(u, x)\}$$

  i.e., based on the d(x) and the weight from x to u to derive d(u)
- Let v is u's parent. If d(u)+w(v,u) > δ → u will be the booster
- If u is a booster, set d(u) to 0 and keep to go upstream nodes


1
2
3
4
5
6
7
8
9
10
4
2
2
1
3
1
4
2
3


1
2^i
3
2^o
5
6^i
6^o
4^i
4^o
9
10
7
8

```
Algorithm TVS(T, δ)
// Determine and output the nodes to be split.
// w() is the weighting function for the edges.
{
    if (T ≠ 0) then
    {
        d[T] := 0;
        for each child v of T do
        {
            TVS(v, δ);
            d[T] := max{d[T], d[v] + w(T, v)};
        }
        if ((T is not the root)  and
                (d[T] + w(parent(T), T) > δ)) then
        {
            write (T); d[T] := 0;
        }
    }
}
```

```
Algorithm TVS(i, δ)
// Determine and output a minimum cardinality split set.
// The tree is realized using the sequential representation.
// Root is at tree[1]. N is the largest number such that
// tree[N] has a tree node.
{
    if (tree[i] ≠ 0) then // If the tree is not empty
        if (2i > N) then d[i] := 0; // i is a leaf.
        else
        {
            TVS(2i, δ);
            d[i] := max(d[i], d[2i] + weight[2i]);
            if (2i + 1 ≤ N) then
            {
                TVS(2i + 1, δ);
                d[i] := max(d[i], d[2i + 1] + weight[2i + 1]);
            }
        }
        if ((tree[i] ≠ 1) and (d[i] + weight[i] > δ)) then
        {
            write (tree[i]); d[i] := 0;
        }
}
```

4.5 Job sequencing with deadline

- Give n jobs. Each job has a deadline d_i and a profit p_i
- If a job can be finished on time, you can earn a reward
- There is a machine. This machine can finish one job per unit of time.
- A feasible solution is that:
  - Find a subset *J* of job and the jobs in subset J can be finished on time (before deadline)
  - Then the profit of the subset J will be
- The target is to fine the schedule that can obtain the maximum profit

Ex 4.5

- Let n=4, (p1,p2,P3,P4) = (100,10,15,27), (d1,d2,d3,d4)=(2,1,2,1)
- The feasible solutions and their values are:

	feasible solution	processing sequence	value
1.	(1, 2)	2, 1	110
2.	(1, 3)	1, 3 or 3, 1	115
3.	(1, 4)	4, 1	127
4.	(2, 3)	2, 3	25
5.	(3, 4)	4, 3	42
6.	(1)	1	100
7.	(2)	2	10
8.	(3)	3	15
9.	(4)	4	27

The greedy policy

- Fine all jobs, and then sort them by their profits
  - Start selecting the one with the most profit
  - Then 2nd profit, and check if feasible …
  - (This fashion may have a drawback that the second one may not be scheduled

  - Another problem is that how to check if feasible →
    - When you select k jobs, you need to check ____ combinations.

  - In actually, you only need to check one possibility
    - Sort the jobs that you selected by their deadline, and then check if exceeding deadline

```
Algorithm GreedyJob(d, J, n)
// J is a set of jobs that can be completed by their deadlines.
{
    J := {1};
    for i := 2 to n do
    {
        if (all jobs in J ∪ {i} can be completed
            by their deadlines) then J := J ∪ {i};
    }
}
```

```
Algorithm JS(d, j, n)
// d[i] ≥ 1, 1 ≤ i ≤ n are the deadlines, n ≥ 1. The jobs
// are ordered such that p[1] ≥ p[2] ≥ ··· ≥ p[n]. J[i]
// is the ith job in the optimal solution, 1 ≤ i ≤ k.
// Also, at termination d[J[i]] ≤ d[J[i + 1]], 1 ≤ i < k.
{
    d[0] := J[0] := 0; // Initialize.
    J[1] := 1; // Include job 1.
    k := 1;
    for i := 2 to n do
    {
        // Consider jobs in nonincreasing order of p[i]. Find
        // position for i and check feasibility of insertion.
        r := k;
        while ((d[J[r]] > d[i]) and (d[J[r]] ≠ r)) do r := r − 1;
        if ((d[J[r]] ≤ d[i]) and (d[i] > r)) then
        {
            // Insert i into J[ ].
            for q := k to (r + 1)  step −1 do J[q + 1] := J[q];
            J[r + 1] := i; k := k + 1;
        }
    }
    return k;
}
```

n=5. (p1,..,p5)=(20,15,10,5,1), (d1,..,d5)=(2,1,1,3,3)

```
Algorithm JS(d, j, n)
// d[i] ≥ 1, 1 ≤ i ≤ n are the deadlines, n ≥ 1. The jobs
// are ordered such that p[1] ≥ p[2] ≥ ··· ≥ p[n]. J[i]
// is the ith job in the optimal solution, 1 ≤ i ≤ k.
// Also, at termination d[J[i]] ≤ d[J[i + 1]], 1 ≤ i < k.
{
    d[0] := J[0] := 0; // Initialize.
    J[1] := 1; // Include job 1.
    k := 1;
    for i := 2 to n do
    {
        // Consider jobs in nonincreasing order of p[i]. Find
        // position for i and check feasibility of insertion.
        r := k;
        while ((d[J[r]] > d[i]) and (d[J[r]] ≠ r)) do r := r − 1;
        if ((d[J[r]] ≤ d[i]) and (d[i] > r)) then
        {
            // Insert i into J[ ].
            for q := k to (r + 1) step −1 do J[q + 1] := J[q];
            J[r + 1] := i; k := k + 1;
        }
    }
    return k;
}
```

To reduce complexity

- Use the union and find operation

- When scheduling a job, try to defer the job according to its deadline
- For a scheduled job i, the algorithm maintains the empty slot in front of i.

```
Algorithm FJS(d, n, b, j)
// Find an optimal solution J[1 : k]. It is assumed that
// p[1] ≥ p[2] ≥ ··· ≥ p[n] and that b = min{n, max_i(d[i])}.
{
    // Initially there are b + 1 single node trees.
    for i := 0 to b do f[i] := i;
    k := 0; // Initialize.
    for i := 1 to n do
    { // Use greedy rule.
        q := CollapsingFind(min(n, d[i]));
        if (f[q] ≠ 0) then
        {
            k := k + 1; J[k] := i; // Select job i.
            m := CollapsingFind(f[q] − 1);
            WeightedUnion(m, q);
            f[q] := f[m]; // q may be new root.
        }
    }
}
```

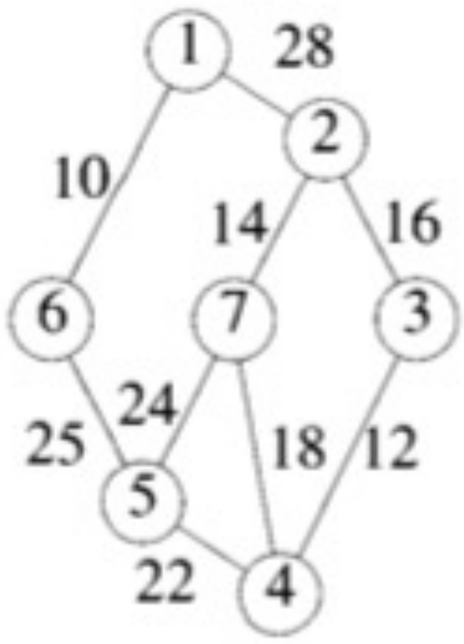
4.6 minimum cost spanning tree

- Give an undirected graph G=(V, E)
- A subgraph t=(V, E’) is a spanning tree iff t is a tree
  - → A spanning tree is a subgraph, which can connect all vertices without a loop

- Assume that each edge has a weight value. The objective function of the MST problem is to find a spanning tree with minimum total weight

Prim’s algorithm

- Find a edge, which can increase the least cost and can also expand the tree

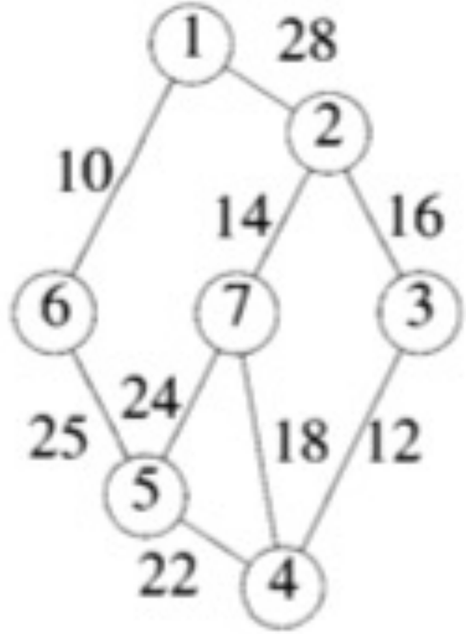


- In actually, you can start anywhere

```
Algorithm Prim(E, cost, n, t)
// E is the set of edges in G. cost[1 : n, 1 : n] is the cost
// adjacency matrix of an n vertex graph such that cost[i, j] is
// either a positive real number or ∞ if no edge (i, j) exists.
// A minimum spanning tree is computed and stored as a set of
// edges in the array t[1 : n − 1, 1 : 2]. (t[i, 1], t[i, 2]) is an edge in
// the minimum-cost spanning tree. The final cost is returned.
{
    Let (k, l) be an edge of minimum cost in E;
    mincost := cost[k, l];
    t[1, 1] := k; t[1, 2] := l;
    for i := 1 to n do  // Initialize near.
        if (cost[i, l] < cost[i, k]) then near[i] := l;
        else near[i] := k;
    near[k] := near[l] := 0;
    for i := 2 to n − 1 do
    { // Find n − 2 additional edges for t.
        Let j be an index such that near[j] ≠ 0 and
        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 to n do // Update near[ ].
            if ((near[k] ≠ 0) and (cost[k, near[k]] > cost[k, j]))
                then near[k] := j;
    }
    return mincost;
}
```

Kruskal’s

- Sort weights by increasing order
- Check in-sequence → if an edge can generate a loop-free spanning tree
  - If yes, select the edge
  - If no, check the next



Kruskal’s

- How to check loop?
  - By concept of set.
    - If some vertices are connected component → put them is a set
    - If a new edge can connect two vertices that are located in different set → OK

```
Algorithm Kruskal(E, cost, n, t)
// 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 edges in the minimum-cost
// spanning tree. The final cost is returned.
{
    Construct a heap out of the edge costs using Heapify;
    for i := 1 to n do parent[i] := -1;
    // Each vertex is in a different set.
    i := 0; mincost := 0.0;
    while ((i < n - 1)  and (heap not empty)) do
    {
        Delete a minimum cost edge (u, v) from the heap
        and reheapify using Adjust;
        j := Find(u); k := Find(v);
        if (j ≠ k) then
        {
            i := i + 1;
            t[i, 1] := u; t[i, 2] := v;
            mincost := mincost + cost[u, v];
            Union(j, k);
        }
    }
    if (i ≠ n - 1) then write ("No spanning tree");
    else return mincost;
}
```

4.7 optimal storage on tapes

- Give n programs and these program should be stored in tapes
- The length of a program i is l_i
- If a programmer wants to access a program inside the tape, he needs to access from the beginning of the tape
- If the sequence of programs are $I = i_1, i_2, \dots, i_n$
- The time to access a program with sequence j will be
- For the storage policy, we need to find a sequence to let the total access time to be minimum

Example 4.8 Let $n = 3$ and $(l_1, l_2, l_3) = (5, 10, 3)$. There are $n! = 6$ possible orderings. These orderings and their respective d values are:

ordering I	$d(I)$		
$1, 2, 3$	$5 + 5 + 10 + 5 + 10 + 3$	$=$	38
$1, 3, 2$	$5 + 5 + 3 + 5 + 3 + 10$	$=$	31
$2, 1, 3$	$10 + 10 + 5 + 10 + 5 + 3$	$=$	43
$2, 3, 1$	$10 + 10 + 3 + 10 + 3 + 5$	$=$	41
$3, 1, 2$	$3 + 3 + 5 + 3 + 5 + 10$	$=$	29
$3, 2, 1$	$3 + 3 + 10 + 3 + 10 + 5$	$=$	34

The optimal ordering is $3, 1, 2$. □

policy

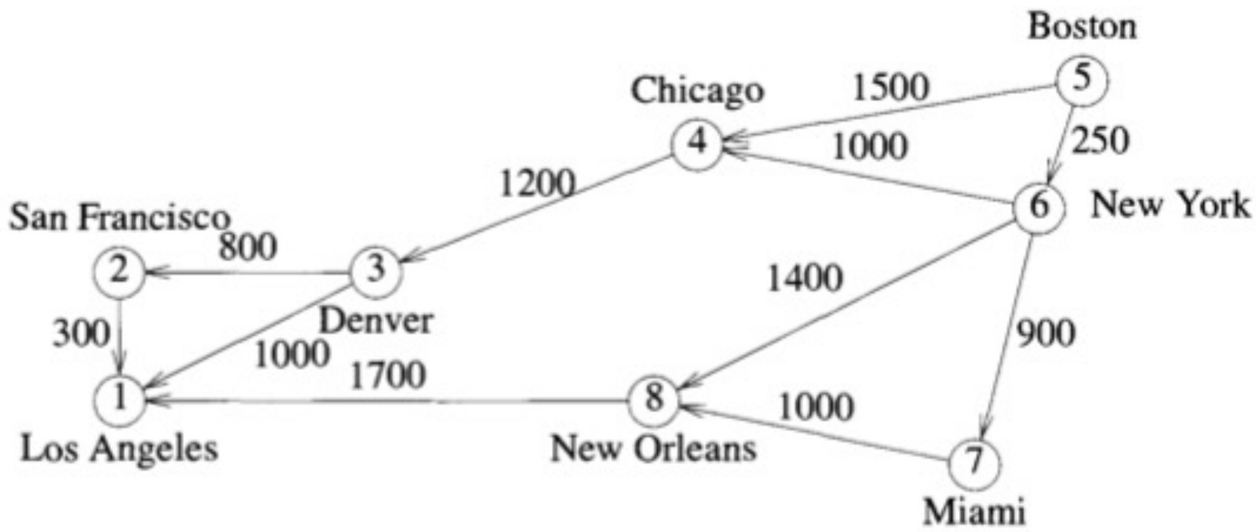
- Let the redundant access time for each program is the least
- Sort programs by their length → shortest job first

- Can extend to multiple tapes (assume that there are m tapes)
  - Sort programs by their lengths again
  - Put first m programs to m tapes, and then follow the operation

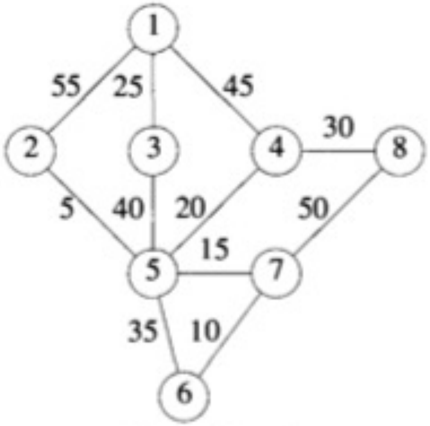
```
Algorithm Store(n, m)
// n is the number of programs and m the number of tapes.
{
    j := 0; // Next tape to store on
    for i := 1 to n do
    {
        write ("append program", i,
            "to permutation for tape", j);
        j := (j + 1) mod m;
    }
}
```

4.9 single source shortest path

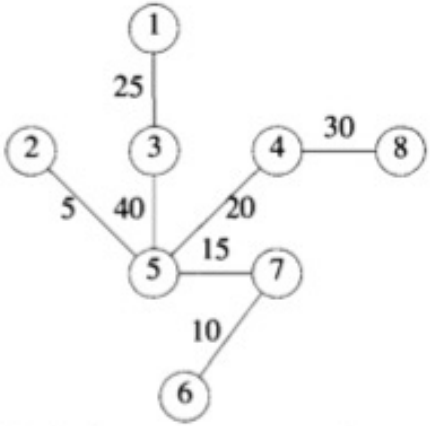
- Give a directed graph G=(V,E)
- Each edge has a weight value, and the weight values are positive
- Target: to find shortest paths from v_0 to all vertices
- Policy: (Dijkstra algorithm)
  - From v_0, find a vertex which has a smallest weight to v_0
  - From the formed subgraph, find a vertex which can have the smallest total weight to v_0 (this is because that the formed paths are guaranteed to be shortest)


Boston
5
Chicago
1500
4
250
1000
1200
6
New York
San Francisco
800
2
3
1400
300
Denver
900
1000
1700
1
8
1000
Los Angeles
New Orleans
7
Miami

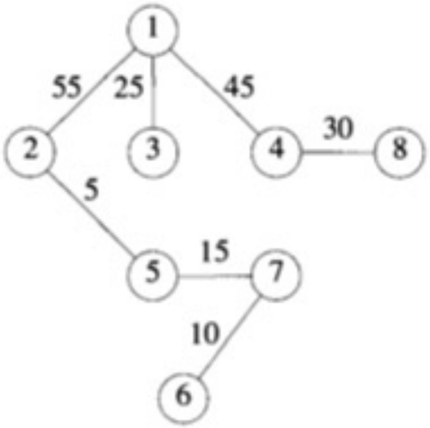
```
Algorithm ShortestPaths(v, cost, dist, n)
// dist[j], 1 ≤ j ≤ n, is set to the length of the shortest
// path from vertex v to vertex j in a digraph G with n
// vertices. dist[v] is set to zero. G is represented by its
// cost adjacency matrix cost[1 : n, 1 : n].
{
    for i := 1 to n do
    { // Initialize S.
        S[i] := false; dist[i] := cost[v, i];
    }
    S[v] := true; dist[v] := 0.0; // Put v in S.
    for num := 2 to n − 1 do
    {
        // Determine n − 1 paths from v.
        Choose u from among those vertices not
        in S such that dist[u] is minimum;
        S[u] := true; // Put u in S.
        for (each w adjacent to u with S[w] = false) do
            // Update distances.
            if (dist[w] > dist[u] + cost[u, w])) then
                    dist[w] := dist[u] + cost[u, w];
    }
}
```



(a) A Graph



(b) Minimum cost spanning tree



(c) Shortest path spanning tree from vertex 1.