

## Module II

### The Greedy Method and Dynamic Programming

The greedy method is the most straight forward algorithm design technique in which there are  $n$  inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies these constraints is called a feasible solution. We need to find a feasible solution that either maximizes or minimizes a given objective function. A feasible solution that does this is called an optimal solution.

Greedy method works in stages, one input at a time in each stages and decision is made based on whether a particular input is in an optimal solution. This is done by considering the inputs in an order determined by some selection procedure. If inclusion of inputs leads to infeasible solution then input is not added to the partial solution. The selection procedure is based on some optimization measure. This measure may be the objective function. Sometimes algorithms that generate suboptimal solutions, this technique is called subset paradigm. For the problems that do not call for the selection of an optimal subset, in the greedy method we make decisions by considering the inputs in some order. This is called ordering paradigm. Following algorithm shows the control abstraction of greedy method.

1. Algorithm Greedy( $a, n$ )
2.  $//a[1:n]$  contains the  $n$  inputs.
3. {
4. solution :=  $\square$ ;  $//$  Initialize the solution
5. for  $i := 1$  to  $n$  do
6. {
7.  $x := \text{Select}(a)$ ;
8. if Feasible(solution,  $x$ ) then
9. solution := Union(solution,  $x$ );
10. }
11. Return solution;
12. }

The function 'select' selects an input from  $a[]$  and removes it. The selected input value is assigned to  $x$ . Feasible is a Boolean valued function that determines whether  $x$  can be included in to the function 'Union' combines  $x$  with the solution and updates the objective function. The function Greedy describes the essential way that a greedy algorithm will look, once a particular problem is chosen and the functions select, feasible and Union are properly implemented.

### Knapsack Problem

There are  $n$  objects and a knapsack or bag. Object  $i$  has a weight  $w_i$  and the knapsack has the capacity  $m$ . If a fraction  $x_i$ ,  $0 \leq x_i \leq 1$ , of objects  $i$  is placed into the knapsack, then a profit of  $p_i x_i$  is earned. The objective is to obtain a filling of the knapsack that maximizes the total

profit earned. Since the knapsack capacity is  $m$ , we require the total weight of all chosen objects to be at most  $m$ .

Formally the problem can be stated as

$$\text{Maximize } \sum p_i x_i, 1 \leq i \leq n \dots \dots \dots (1)$$

$$\text{Subject to } \sum w_i x_i \leq m, 1 \leq i \leq n \dots \dots \dots (2)$$

$$\text{And } 0 \leq x_i \leq 1, 1 \leq i \leq n \dots \dots \dots (3)$$

The profit and weights are positive numbers. A feasible solution is any set  $(x_1, x_2, \dots, x_n)$  satisfying (2) and (3). An optimal solution is feasible solution for which (1) is maximized.

They are two types of knapsack problem

- 1) **0-1 knapsack problem:** Here the items may not be broken into smaller pieces, so it may decide either to enter an item into the bag or leave it (binary choice). It cannot be efficiently solved by greedy algorithm.
- 2) **Fractional Knapsack problem:** Here it can take the fraction of items, meaning that the items can be broken into smaller pieces so that it may be able to carry a fraction  $x_i$  of item  $i$ . This can be solved easily by greedy.

Eg; consider the following instance of the knapsack problem.

$n=3, m=20$ ,  $(P_1, P_2, P_3) = (25, 24, 15)$  &  $(w_1, w_2, w_3) = (18, 15, 10)$

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

Note that knapsack problem calls for select a subset of the objects hence fits the subset paradigm.

In first method, we can try to fill the knapsack by including the object with largest profit (greedy approach to the profit). If an object under consideration does not fit, then a fraction of it is included to fit the knapsack. Object 1 has the largest profit value,  $P_1=25$ . So it is placed into the knapsack first. Then  $x_1=1$  and a profit of 25 is earned. Only 2 units of knapsack capacity are left. Object 2 has the next largest profit  $P_2=24$ . But  $W_2=15$  & it does not fit into the knapsack. Using  $x_2 = 2/15$  fills the knapsack exactly with the part of the object 2.

The method used to obtain this solution is termed a greedy method at each step, we chose to introduce that object which would increase the objective function value the most.

$$(x_1, x_2, x_3) \quad \sum w_i x_i \quad \sum p_i x_i$$

$$(1, 2/15, 0) \quad 20 \quad 28.2$$

This is not an optimal solution.

In another method we apply greedy approach by choosing value per unit weight is as high as possible

Item(n)	Value(p1,p2,p3)	Weight(w1,w2,w3)	Val/weight
1	25	18	1.388
2	24	15	1.6
3	15	10	1.5

Here  $p_2/w_2 > p_3/w_3 > p_1/w_1$ . Now the items are arranged into non increasing order of  $p_i/w_i$ . Second item is the most valuable item. We chose item 2 first. Item 3 is second most valuable item. But we cannot choose the entire item3 as the weight of item 3 exceeds the capacity of knapsack . We can take . of the third item. Therefore the solution is  $x_1 = 0$  ,  $x_2 = 1$  ,  $x_3 = .$  and maximum profit is

$$\sum p_i x_i = 0*25 + 1*24 + . * 15 = 31.5$$

If the items are already arranged in non increasing order of  $p_i/w_i$  , then the function greedyknapsack obtains solution corresponding to this strategy.

1. Algorithm greedyknapsack (m,n)
2. // p [1:n] and w[1:n] contain the profits and weights respectively
3. // of the n objects ordered such that  $p[i]/w[i] \geq p[i+1]/w[i+1]$
4. // m is the knapsack size and x[1:n] is the solution vector
5. {
6. for i := 1 to n do x[i] = 0;
7. U:= m;
8. for i := 1 to n do
9. {
10. if ( w(i) > U) then break;
11. x[i]:=1.0; U:=U - w[i];
12. }
13. if (i ≤ n) then x[i] := U/w[i];
14. }

## Analysis

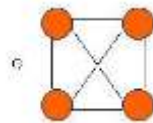
If the items are already sorted into decreasing order of  $p_i/w_i$ , then time complexity is  $O(n)$   
Therefore Time complexity including sort is  $O(n \log n)$

## Minimum-Cost Spanning Tree

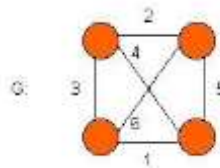
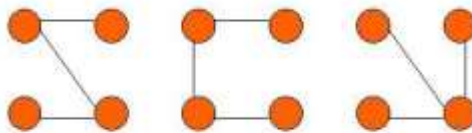
A tree is defined to be an undirected, acyclic and connected graph (or more simply, a graph in which there is only one path connecting each pair of vertices). Assume there is an undirected, connected graph  $G$ . A spanning tree is a subgraph of  $G$ , is a tree, and contains all the vertices of  $G$ . A minimum spanning tree is a spanning tree, but has weights or lengths associated with the edges, and the total weight of the tree (the sum of the weights of its edges) is at a minimum.

### Application of MST

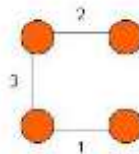
- 1) practical application of a MST would be in the design of a network. For instance, a group of individuals, who are separated by varying distances, wish to be connected together in a telephone network. MST can be used to determine the least costly paths with no cycles in this network, thereby connecting everyone at a minimum cost.
- 2) Another useful application of MST would be finding airline routes. MST can be applied to optimize airline routes by finding the least costly paths with no cycles.



Three spanning trees from graph  $G$ .



A weighted graph  $G$



The minimum spanning tree from weighted graph  $G$

### Prim's algorithm

It is an algorithm in graph theory that 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

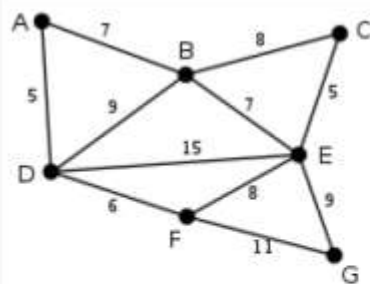
scientist Robert C. Prim in 1957 and rediscovered by Edsger Dijkstra in 1959. Therefore it is sometimes called the DJP algorithm, the Jarnik algorithm, or the Prim-Jarnik algorithm.

### **How it works:**

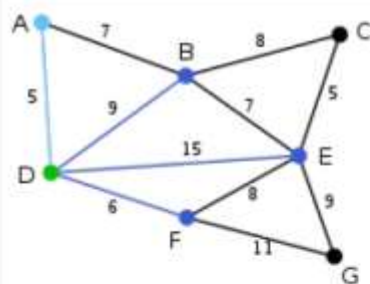
1. This algorithm builds the MST one vertex at a time.
2. It starts at any vertex in a graph (vertex A, for example), and finds the least cost vertex (vertex B, for example) connected to the start vertex.
3. Now, from either 'A' or 'B', it will find the next least costly vertex connection, without creating a cycle (vertex C, for example).
4. Now, from either 'A', 'B', or 'C', it will find the next least costly vertex connection, without creating a cycle, and so on it goes.
5. Eventually, all the vertices will be connected, without any cycles, and an MST will be the result.

Example: find the minimum cost path using prim's Algorithm

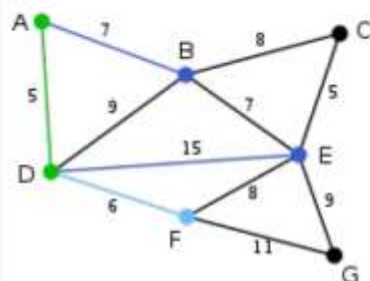
Image	Description
-------	-------------



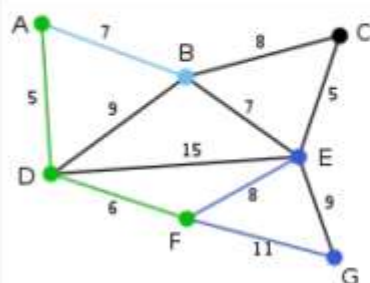
This is our original weighted graph. The numbers near the edges indicate their weight.



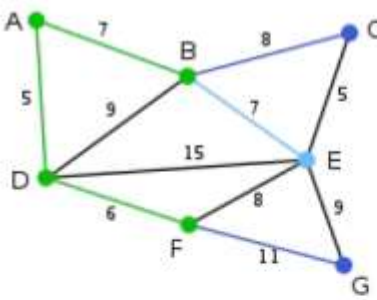
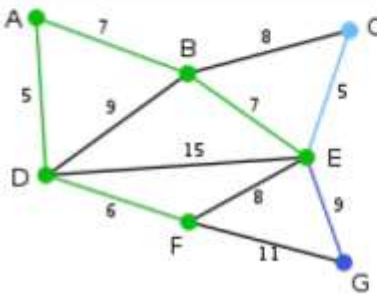
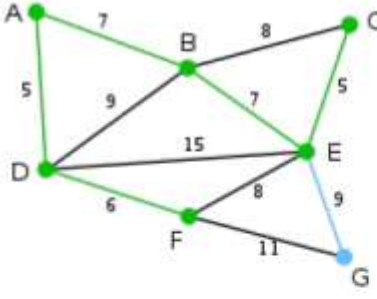
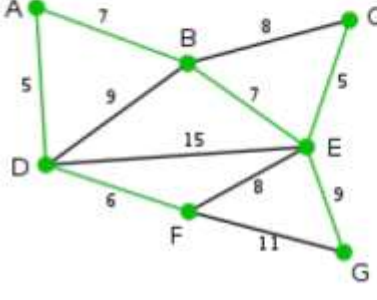
Vertex **D** has been arbitrarily chosen as a starting point. Vertices **A**, **B**, **E** and **F** are connected to **D** through a single edge. **A** is the vertex nearest to **D** and will be chosen as the second vertex along with the edge **AD**.



The next vertex chosen is the vertex nearest to *either* **D** or **A**. **B** is 9 away from **D** and 7 away from **A**, **E** is 15, and **F** is 6. **F** is the smallest distance away, so we highlight the vertex **F** and the arc **DF**.



The algorithm carries on as above. Vertex **B**, which is 7 away from **A**, is highlighted.

	<p>In this case, we can choose between <b>C</b>, <b>E</b>, and <b>G</b>. <b>C</b> is 8 away from <b>B</b>, <b>E</b> is 7 away from <b>B</b>, and <b>G</b> is 11 away from <b>F</b>. <b>E</b> is nearest, so we highlight the vertex <b>E</b> and the arc <b>BE</b>.</p>
	<p>Here, the only vertices available are <b>C</b> and <b>G</b>. <b>C</b> is 5 away from <b>E</b>, and <b>G</b> is 9 away from <b>E</b>. <b>C</b> is chosen, so it is highlighted along with the arc <b>EC</b>.</p>
	<p>Vertex <b>G</b> is the only remaining vertex. It is 11 away from <b>F</b>, and 9 away from <b>E</b>. <b>E</b> is nearer, so we highlight it and the arc <b>EG</b>.</p>
	<p>Now all the vertices have been selected and the <a href="#">minimum spanning tree</a> is shown in green. In this case, it has weight 39</p>

Algorithm

```

1. Algorithm (E, cost, n, t)
2. // E is the set of edges, cost[1: n, 1: n] is the cost.
3. // adjacency matrix of an n vertex graph such that cost[i, j] is
4. // either a positive real number or  $\infty$  if no edge (i, j) exists.
5. // A minimum spanning tree is computed and stored as a set of
6. // edges in the array t[1: n-1, 1:2]. (t[i, 1], t[i, 2]) is an edge in
7. // the minimum-cost spanning tree. The final cost is returned.
8. {
9.   // Let (k, l) be an edge of minimum cost in E;
10.  mincost = cost[k, l];
11.  t[1, 1] := k; t[1, 2] := l;
12.  for i := 1 to n do // initialize near
13.    if (cost[i, l] < cost[i, k]) then near[i] := l;
14.  else near[i] := k;
15.  near[k] := near[l] := 0;
16.  for i := 2 to n-1 do
17.    { // Find n-2 additional edges for t
18.      let j be an index such that near[j]  $\neq$  0 and
19.      cost[j, near[j]] is minimum;
20.      t[i, 1] := j; t[i, 2] := near[j];
21.      mincost := mincost + cost[j, near[j]];
22.      near[j] := 0;
23.      for k := 1 to n do // Update near[]
24.        if ((near[k]  $\neq$  0) and (cost[k, near[k]] > cost[k, j]))
25.          then near[k] := j;
26.    }
27.  return mincost;
28. }
```

The algorithm will start with a tree that includes only a minimum cost edge of G. Then edges are added to this tree one by one. The next edge (i, j) to be added is such that i is a vertex already included in the tree, j is a vertex not included and cost[i, j] is the minimum among all edges. To determine this edge (i, j) efficiently we associate with each vertex 'j' not yet included in the tree, a value near[j]. The value near[j] is a vertex in the tree such that cost[j, near[j]] is minimum among all choices for near[j]. we define near[j] = 0 for all vertices j that are already in the tree. The next edge to include is defined by the vertex j such that near[j]  $\neq$  0 (j not already in the tree) and cost[j, near[j]] is minimum.

### Time complexity

Since we have 2 nested for loops. Therefore overall time complexity is  $O((n+E) \log n)$ .



## Kruskal's algorithm

It is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. Kruskal's algorithm is an example of a greedy algorithm. It works as follows:

- create a forest  $F$  (a set of trees), where each vertex in the graph is a separate tree
- create a set  $S$  containing all the edges in the graph
- while  $S$  is nonempty and  $F$  is not yet spanning
  - remove an edge with minimum weight from  $S$
  - if that edge connects two different trees, then add it to the forest, combining two trees into a single tree
  - otherwise discard that edge.

At the termination of the algorithm, the forest has only one component and forms a minimum spanning tree of the graph.

General form of Kruskal Algorithm

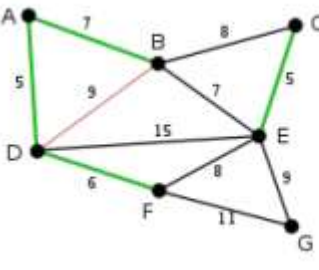
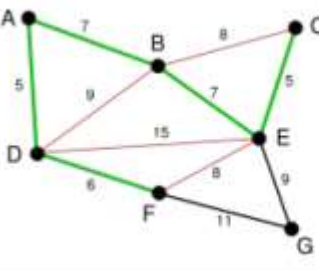
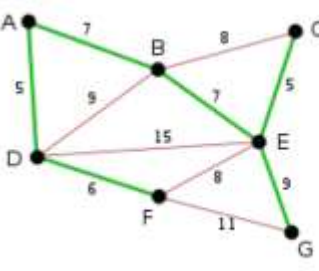
1.  $t = 0$ ; //  $t$  is the minimum spanning tree
2. while (( $t$  has less than  $n-1$  edges) and ( $E \neq \emptyset$ )) do
3. {
4. Choose an edge  $(v,w)$  from  $E$  of lowest cost;
5. Delete  $(v,w)$  from  $E$ ;
6. If  $(v,w)$  does not create a cycle in  $t$  then add  $(v,w)$  to  $t$ ;
7. else discard  $(v,w)$ ;
8. }

Initially  $E$  is the set of all edges in  $G$ . The only functions we wish to perform on this set are

- 1) determine an edge with minimum cost
- 2) delete this edge

Example

Image	Description
	This is our original graph. The numbers near the arcs indicate their weight. None of the arcs are highlighted.
	<b>AD</b> and <b>CE</b> are the shortest arcs, with length 5, and <b>AD</b> has been <u>arbitrarily</u> chosen, so it is highlighted.
	<b>CE</b> is now the shortest arc that does not form a cycle, with length 5, so it is highlighted as the second arc.
	The next arc, <b>DF</b> with length 6, is highlighted using much the same method.

	<p>The next-shortest arcs are <b>AB</b> and <b>BE</b>, both with length 7. <b>AB</b> is chosen arbitrarily, and is highlighted. The arc <b>BD</b> has been highlighted in red, because there already exists a path (in green) between <b>B</b> and <b>D</b>, so it would form a cycle (<b>ABD</b>) if it were chosen.</p>
	<p>The process continues to highlight the next-smallest arc, <b>BE</b> with length 7. Many more arcs are highlighted in red at this stage: <b>BC</b> because it would form the loop <b>BCE</b>, <b>DE</b> because it would form the loop <b>DEBA</b>, and <b>FE</b> because it would form <b>FEBAD</b>.</p>
	<p>Finally, the process finishes with the arc <b>EG</b> of length 9, and the minimum spanning tree is found.</p>

This algorithm first appeared in Proceedings of the American Mathematical Society, 1956, and was written by Joseph Kruskal.

```

1 Algorithm Kruskal( E, cost, n, t)
2 // E is the set of edges in G. G has n vertices. cost[u,v] is the
3 // cost of edge (u,v). t is the set of edges in the minimum cost
4 // spanning tree. The final cost is returned.
5 {
6   construct a heap out of the edge cost using Heapify;
7   for i:= 1 to n do parent[i] := -1;
8   // each vertex is in a different set
9   i := 0; mincost := 0.0 ;
10  while(( i < n-1) and (heap not empty)) do
11  {
12    delete a minimum cost edge (u,v) from the heap
13    and reheapify using adjust;
14    j:= Find(u); k := Find(v);
15    if ( j ≠ k) then

```

```

16 {
17 i:=i+1;
18 t[i,1] :=u; t[i,2]:= v;
19 mincost := mincost + cost[u,v];
20 Union(j,k);
21 }
22 }
23 if (I ≠ n-1) then write (“no spanning tree”);
24 else return mincost;
}

```

### Explanation

- t is the set of edge to be included in the minimum cost spanning tree
- i is the number of edges in the graph
- Edge (u,v) can be added to t by the assignments  $t[i,1] := u$  and  $t[i,2] := v$ ;
- In the while loop edges are removed from the heap one by one in increasing order of the
- cost
- $j = \text{find}(u)$ ,  $k = \text{find}(v)$  determines the sets containing u and v.
- If  $j \neq k$ , then vertices u and v are in different sets and edge (u,v) is included into t.
- If  $j=k$ , the edge (u,v) is discarded as its inclusion into t would create a cycle.
- Union(j,k) combines the set containing u and v
- Determines whether a spanning tree was found. It follows that  $i \neq n-1$  iff the graph G is not connected.
- The computing time is  $O(E \log E)$  where E is the edge set of G

### Job sequencing with deadlines

We are given a set of n jobs. Associated with job i is an integer deadline  $d_i \geq 0$  and a profit  $P_i \geq 0$ . For any job i profit  $P_i$  is earned iff the job is completed by its deadline. To complete a job, one has to process job on a machine for one unit of time. Only one machine is available for processing jobs. A feasible solution for this problem is a subset J of jobs such that each job in this subset can be completed by its deadline. The value of a feasible solution J is the sum of the profits of the jobs in J, or  $\sum P_i$ . An optimal solution is a feasible solution with maximum value. Here the problem involves the identification of a subset, it fits the subset paradigm.

Eg; Let  $n=4$ .  $(P_1, P_2, P_3, P_4) = (100, 10, 15, 27)$  and  $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$ .  $d_1 = 2$  means first job should be completed by first 2 units of time.  $d_2 = 1$  means second job should be completed by first 1 unit of time. 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

Solution 3 is optimal. In this solution job 1 & 4 are produced and the value is 127. These jobs must be processed in the order job 4 followed by job 1. Thus the processing of job 4 begins at time zero & that of job 1 is completed at time 2. We can choose the objective function  $\sum P_i, i \in J$  as one optimization measure. Using this measure, the next job to include is the one that increases  $\sum P_i, i \in J$ , the most, subject to the constraint that the resulting  $J$  is a feasible solution. This requires us to consider jobs in decreasing order of  $P_i$ 's.

From the above example we begin with  $J=0$  (null) and  $\sum P_i = 0, i \in J$ . Jobs 1 is added to  $J$  as it has the largest profit and  $J=\{1\}$  is a feasible solution. Next job 4 is considered. The solution  $J=\{1,4\}$  is also feasible. Next job 3 is considered and discarded as  $J=\{1,3,4\}$  is not feasible. Hence we are left with the solution.  $J=\{1,4\}$  with value 127. This is the optimal solution for the problem instance.

1. Algorithm GreedyJob(d, J, n)
2. //J is a set of jobs that can be completed by their deadlines.
3. {
4.  $J := \{1\};$
5. for  $i := 2$  to  $n$  do
6. {
7.     If (all jobs in  $J \cup \{i\}$  can be completed
8.     by their deadlines) then  $J := J \cup \{i\};$
9. }
10. }

## DYNAMIC PROGRAMMING

In dynamic programming the sub problem is solved only once and the answer saved in a table and solution to a problem can be viewed as the result of a sequence of decisions.

### Eg. 1: Knapsack problem solution

- Decide the values of  $x_i, 0 \leq i < n$
- Make a decision on  $x_0$ , then  $x_1$ , and so on
- An optimal sequence maximizes the objective function  $\sum p_i x_i$  under the constraints  $\sum w_i x_i \leq m$  and  $0 \leq x_i \leq 1$

### Eg. 2: Shortest path problem

- Determine the shortest path from vertex  $i$  to vertex  $j$  by finding the second vertex in the path, then the third, and so on, until vertex  $j$  is reached.
- Optimal sequence of decisions is one with a path of least length

Dynamic programming drastically reduces the amount of enumeration by avoiding the enumeration of some decision sequences that cannot possibly be optimal; an optimal sequence of decisions is obtained by using the *principle of optimality*.

- Applicable when the subproblems are not entirely independent, they may have common subproblems
- Each subsubproblem is solved only once and the results used repeatedly

The word *programming* in dynamic programming does not refer to coding but refers to building tables of intermediate results.

The **principle of optimality** states that an optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining states must constitute an optimal decision sequence with regard to the state resulting from the first decision.

### Greedy method vs Dynamic programming

- In greedy method, only one decision sequence is ever generated

### Dynamic Programming

- In dynamic programming, many decision sequences may be generated
- Sequences containing suboptimal sequences cannot be optimal because of principle of optimality, and so, will not be generated

One of the features of dynamic programming is algorithms often have a polynomial complexity. This is because the total number of different decision sequences is exponential in the number of decisions (if there are  $d$  choices for each of the  $n$  decisions to be made then there are  $d^n$  possible decision sequences.)

Another important feature of the dynamic programming approach is that optimal solutions to subproblems are retained so as to avoid recomputing their values. The use of these tabulated values makes it natural to recast the recursive equations into an iterative algorithm.

## The Traveling Sales Person Problem

Let  $G=(V,E)$  be a directed graph with edge cost  $c_{ij}$ . The variable  $c_{ij}$  is defined such that  $c_{ij}>0$  for all  $i$  and  $j$  and  $c_{ij} = \infty$  if  $\langle i, j \rangle \notin E$ . Let  $|V|=n$  and assume  $n > 1$ . A tour of  $G$  is a directed simple cycle that includes every vertex in  $V$ . The cost of a tour is the sum of the cost of the edges on the tour. The traveling salesperson problem is to find a tour of minimum cost.

The traveling sales person problem finds application in a variety of situations. Suppose we have to route a postal van to pick up mail from mail boxes located at  $n$  different sites. An  $n + 1$  vertex graph can be used to represent the situation. One vertex represents the post office from which the postal van starts and to which it must return. Edge  $\langle i, j \rangle$  is assigned a cost equal to the distance from site  $i$  to site  $j$ . The route taken by the postal van is a tour, and we are interested in finding a tour of minimum length.

Egs.

- We wish to use a robot arm to tighten the nuts on some piece of machinery on an assembly line.
- In a production environment, several commodities are manufactured on the same set of machines.

It can say a tour to be a simple path that starts and ends at vertex 1. Every tour consists of an edge  $\langle 1, k \rangle$  for some  $k \in V - \{1\}$  and a path from vertex  $k$  to vertex 1. The path from vertex  $k$  to vertex 1 goes through each vertex in  $V - \{1, k\}$  exactly once. It is easy to see that if the tour is optimal, then the path from  $k$  to 1 must be a shortest  $k$  to 1 path going through all vertices in  $V - \{1, k\}$ . Hence, the principle of optimality holds. Let  $g(i, S)$  be the length of a shortest path starting at vertex  $i$ , going through all vertices in  $V - \{1, k\}$ . hence, the principle of optimality holds. Let  $g(i, S)$  be the length of a shortest path starting at vertex  $i$ , going through all vertices in  $S$ , and terminating at vertex 1. The function  $g(1, V - \{1\})$  is the length of an optimal salesperson tour. From the principle of optimality it follows that

$$g(1, V - \{1\}) = \min_{2 \leq k \leq n} \{c_{1k} + g(k, V - \{1, k\})\} \quad (5.20)$$

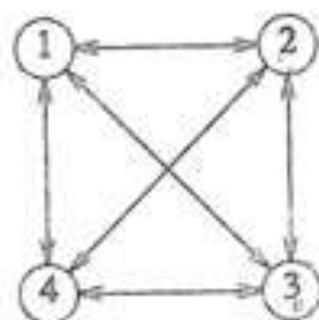
Generalizing (5.20), we obtain (for  $i \notin S$ )

$$g(i, S) = \min_{j \in S} \{c_{ij} + g(j, S - \{j\})\} \quad (5.21)$$

Equation 5.20 can be solved for  $g(1, V - \{1\})$  if we know  $g(k, V - \{1, k\})$  for all choices of  $k$ . the  $g$  values can be obtained by using 5.21. Clearly

$g(i, \phi) = c_{i1}$ ,  $1 \leq i \leq n$ . Hence, we can use (5.21) to obtain  $g(i, S)$  for  $S$  of size 1. Then we can obtain  $g(i, S)$  for  $S$  with  $|S| = 2$ , and so on. When  $|S| < n - 1$ , the values of  $i$  and  $S$  for which  $g(i, S)$  is needed are such that  $i \neq 1$ ,  $1 \notin S$ , and  $i \notin S$ .

**Example 5.26** Consider the directed graph of Figure 5.21(a). The edge lengths are given by matrix  $c$  of Figure 5.21(b).



(a)

0	10	15	20
5	0	9	10
6	13	0	12
8	8	9	0

(b)

**Figure 5.21** Directed graph and edge length matrix  $c$

Thus  $g(2, \phi) = c_{21} = 5$ ,  $g(3, \phi) = c_{31} = 6$ , and  $g(4, \phi) = c_{41} = 8$ . Using (5.21), we obtain

$$\begin{aligned} g(2, \{3\}) &= c_{23} + g(3, \phi) = 15 & g(2, \{4\}) &= 18 \\ g(3, \{2\}) &= 18 & g(3, \{4\}) &= 20 \\ g(4, \{2\}) &= 13 & g(4, \{3\}) &= 15 \end{aligned}$$

Next, we compute  $g(i, S)$  with  $|S| = 2$ ,  $i \neq 1$ ,  $1 \notin S$  and  $i \notin S$ .

$$\begin{aligned} g(2, \{3, 4\}) &= \min \{c_{23} + g(3, \{4\}), c_{24} + g(4, \{3\})\} = 25 \\ g(3, \{2, 4\}) &= \min \{c_{32} + g(2, \{4\}), c_{34} + g(4, \{2\})\} = 25 \\ g(4, \{2, 3\}) &= \min \{c_{42} + g(2, \{3\}), c_{43} + g(3, \{2\})\} = 23 \end{aligned}$$

Finally, from (5.20) we obtain

$$\begin{aligned} g(1, \{2, 3, 4\}) &= \min \{c_{12} + g(2, \{3, 4\}), c_{13} + g(3, \{2, 4\}), c_{14} + g(4, \{2, 3\})\} \\ &= \min \{35, 40, 43\} \\ &= 35 \end{aligned}$$



The optimal tour of the graph has length 35. Let  $J(i, S)$  be this value. Then  $J(1, \{2,3,4\})=2$ . Thus the tour starts from 1 and goes to 2. The remaining tour can be obtained from  $g(2, \{3,4\})$ . So  $g(2, \{3,4\}) = 4$ . Thus the next edge is  $\langle 2, 4 \rangle$ . The remaining tour is for  $g(4, \{3\})$ . So  $J(4, \{3\}) = 3$ . The optimal tour is 1, 2, 4, 3, 1.

Let  $N$  be the number of  $g(i, S)$ 's that have to be computed before 5.20 can be used to compute  $g(1, V - \{1\})$ . For each value of  $|S|$  there are  $n-1$  choices for  $i$ . the number of distinct sets  $S$  of size  $k$  not including 1 and  $i$  is

$$\binom{n-2}{k}. \text{ Hence}$$

$$N = \sum_{k=0}^{n-2} (n-1) \binom{n-2}{k} = (n-1)2^{n-2}$$

Algorithm to find optimal tour takes  $\Theta(n^2 2^n)$ . The most serious drawback of this dynamic programming solution is the space needed,  $O(n 2^n)$ . This is too large even for modest values of  $n$ .

### All-pairs Shortest Paths

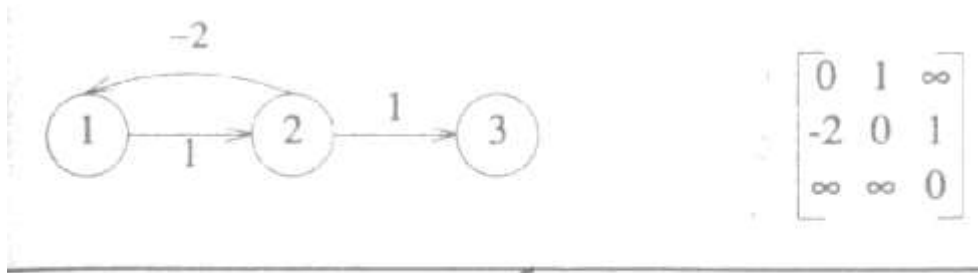
Let  $G=(V, E)$  be a directed graph with  $n$  vertices. Let cost be a cost adjacency matrix for  $G$  such that  $\text{cost}(i, i)=0, 1 \leq i \leq n$ . Then  $\text{cost}(i, j)$  is the length (cost) of edge  $\langle i, j \rangle$  if  $\langle i, j \rangle \in E(G)$  and  $\text{cost}(i, j) = \infty$  if  $i \neq j$  and  $\langle i, j \rangle \notin E(G)$ . The all pairs shortest-path problem is to determine a matrix  $A$  such that  $A(i, j)$  is the length of a shortest path from  $i$  to  $j$ . It is possible to obtain a solution with  $O(n^3)$  time using the principle of optimality.

For every edge  $\langle i, j \rangle$  we require that  $G$  have no cycles with negative length. If we allow  $G$  to contain a cycle of negative length, then the shortest path between any two vertices on this cycle has length  $-\infty$ . A shortest  $i$  to  $j$  path in  $G, i \neq j$  originates at vertex  $i$  and goes through some intermediate vertices and terminates at vertex  $j$ . If  $k$  is an intermediate vertex on this shortest path, then the subpaths from  $i$  to  $k$  and from  $k$  to  $j$  must be shortest paths from  $i$  to  $k$  and  $k$  to  $j$ , respectively. Otherwise, the  $i$  to  $j$  path is not of minimum length. So the principle of optimality holds.

If  $k$  is the intermediate vertex with highest index, then the  $i$  to  $k$  path is a shortest  $i$  to  $k$  path in  $G$  going through no vertex with index greater than  $k-1$ .

$$A^k(i, j) = \min \{ A^{k-1}(i, j), A^{k-1}(i, k) + A^{k-1}(k, j) \}, k \geq 1.$$

The following example shows that above equation is not true for graphs with cycles of negative length.



Graph with negative cycle

For this graph  $A^2(1, 3) \neq \min \{ A^1(1,3), A^1(1, 2) + A^1(2,3) \} = 2$ . Instead we see that  $A^2(1, 3) = -\infty$ . The length of the path 1, 2, 1, 2, 1, 2, ..., 1, 2, 3. This is because of presence of the cycle 1 2 1 which has a length of -1.

Recurrence can be solved for  $A^n$  by first computing  $A^1$ , then  $A^2$ , and so on. Since there is no vertex in  $G$  with index greater than  $n$ ,  $A(i, j) = A^n(i, j)$ .

The following graph has the cost matrix. The initial  $A^{(0)}$ , plus its values after 3 iterations  $A^{(1)}$ ,  $A^{(2)}$ ,  $A^{(3)}$  and given in the figure.

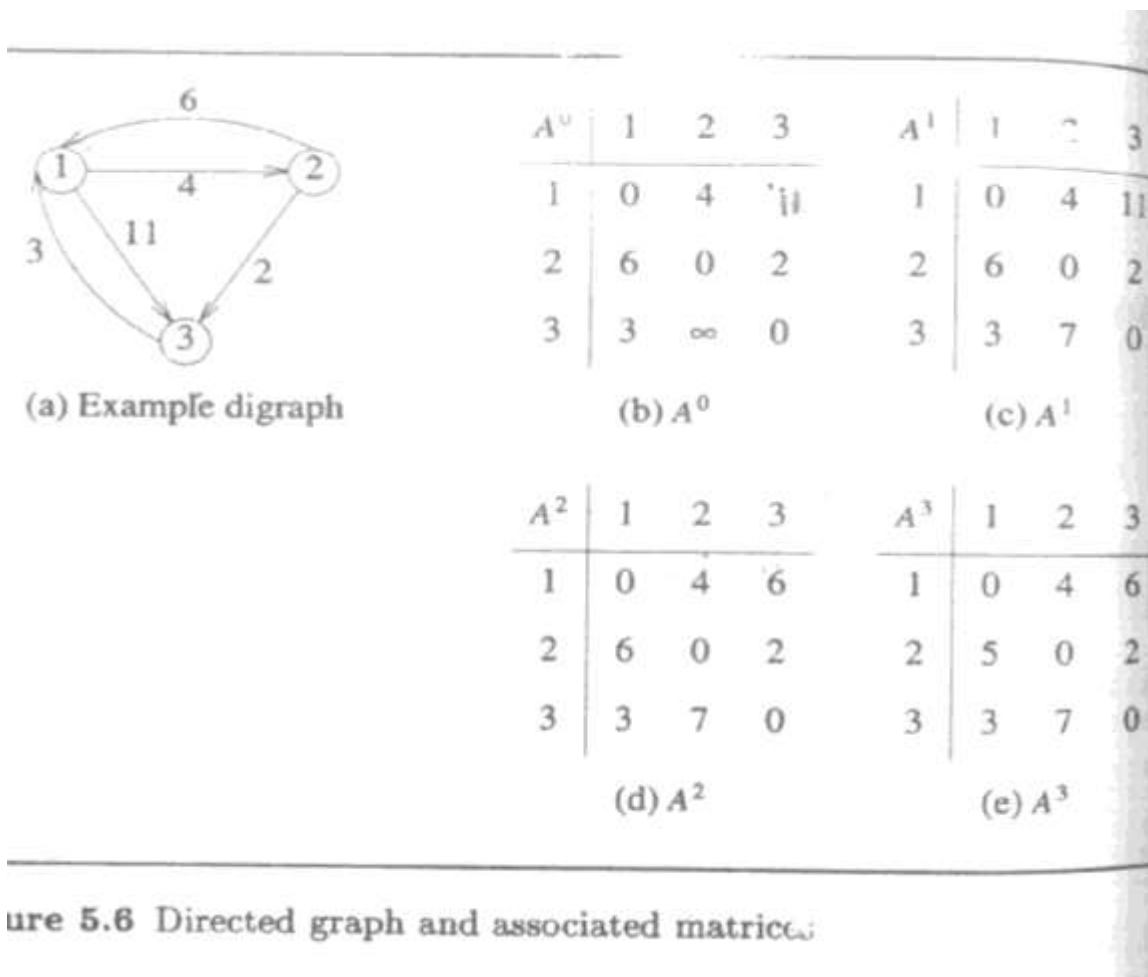


Figure 5.6 Directed graph and associated matrices

```

1. Algorithm AllPths(cost, A, n)
2. // cost[1:n, 1:n] is the cost adjacency matrix of a graph with
3. // n vertices; A[i, j] is the cost of a shortest path from vertex
4. // i to vertex j. cost[i, j] = 0.0 for  $1 \leq i \leq n$ .
5. {
6.   for i:= 1 to n do
7.     for j:= 1 to n do
8.       A[i, j]:= cost [i, j]; // Copy cost into A.
9.   for k:=1 to n do
10.    for i:= 1 to n do
11.      for j:= 1 to n do
12.        A[i, j]:= min(A[i, j], A[i,k] + A[k, j]);
13. }

```

The time needed by AllPaths()  $\Theta(n^3)$  because the line 12 is iterated  $n^3$  times.

### Single source shortest paths : General Weights

Some or all the edges of directed graph  $G$  may have negative length. When negative edge lengths are permitted, we require that the graph have no cycles of negative length. This is necessary to ensure that shortest paths consists of a finite number of edges.

Let  $\text{dist}^l[u]$  be the length of a shortest path from the source vertex  $v$  to vertex  $u$  under the constraint that the shortest path contains at most  $l$  edges. Then  $\text{dist}^1[u] = \text{cost}[v, u]$ ,  $1 \leq u \leq n$ .  $\text{dist}^{n-1}[u]$  is the length of an unrestricted shortest path from  $v$  to  $u$ . This problem is to compute  $\text{dist}^{n-1}[u]$  for all  $u$ . this can be done using the dynamic programming methodology. Following recurrence can be used for dist:

$$\text{dist}^k[u] = \min \{ \text{dist}^{k-1}[u], \min (\text{dist}^{k-1}[i] + \text{cost}[i, u]) \}$$

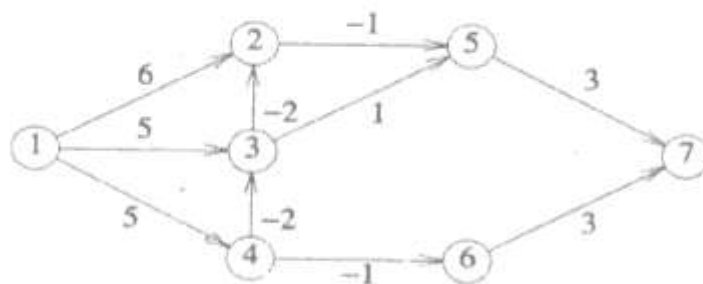
This recurrence can be used to compute  $\text{dist}^k[u]$  from  $\text{dist}^{k-1}$  for  $k = 2, 3, \dots, n-1$

Eg. Following graph shows the paths with negative edge lengths. It is a seven vertex graph, together with arrays  $k=1, 2, \dots, 6$ . These arrays can be computed using equation above. For instance  $\text{dist}[1]=0$   $\text{dist}[2]=6$ ,  $\text{dist}[3]=5$  etc.

1 to these nodes. The distance  $dist^1[]$  is  $\infty$  for the nodes 5, 6, and 7 since there are no edges to these from 1.

$$\begin{aligned} dist^2[2] &= \min \{ dist^1[2], \min_i \{ dist^1[i] + cost[i, 2] \} \} \\ &= \min \{ 6, 0 + 6, 5 - 2, 5 + \infty, \infty + \infty, \infty + \infty, \infty + \infty \} = 3 \end{aligned}$$

Here the terms  $0 + 6, 5 - 2, 5 + \infty, \infty + \infty, \infty + \infty$ , and  $\infty + \infty$  correspond to a choice of  $i = 1, 3, 4, 5, 6$ , and 7, respectively. The rest of the entries are computed in an analogous manner.



(a) A directed graph

k	$dist^k[1..7]$						
	1	2	3	4	5	6	7
1	0	6	5	5	$\infty$	$\infty$	$\infty$
2	0	3	3	5	5	4	$\infty$
3	0	1	3	5	2	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

(b)  $dist^k$

Figure 5.10 Shortest paths with negative edge lengths

The algorithm to find shortest path is called Bellman and Ford algorithm. This is given below:

---

```

1  Algorithm BellmanFord(v, cost, dist, n)
2  // Single-source/all-destinations shortest
3  // paths with negative edge costs
4  {
5      for i := 1 to n do // Initialize dist.
6          dist[i] := cost[v, i];
7      for k := 2 to n - 1 do
8          for each u such that u ≠ v and u has
9              at least one incoming edge do
10             for each (i, u) in the graph do
11                 if dist[u] > dist[i] + cost[i, u] then
12                     dist[u] := dist[i] + cost[i, u];
13 }

```

---

**Algorithm 5.4** Bellman and Ford algorithm to compute shortest paths

For loop take  $O(n^2)$  time if the adjacency matrices are used and  $O(e)$  time if adjacency lists are used. Here  $e$  is the number of edges in the graph. The overall complexity is  $O(n^3)$  if adjacency matrices are used and  $O(ne)$  is the time if adjacency lists are used.