3. 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 $p_1/w_1 \ge p_2/w_2 \ge \ldots p_n/w_n$

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

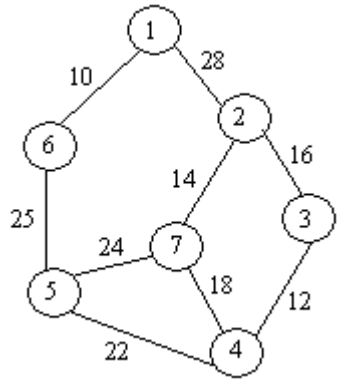
total weight of all the edges in the tree is minimized. The algorithm was discovered in 1930 by mathematician Vojtěch Jarník and later independently by computer scientist Robert C. Prim in 1957 and rediscovered by Edsger Dijkstra in 1959. Therefore it is sometimes called the DJP algorithm, the Jarník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

Steps

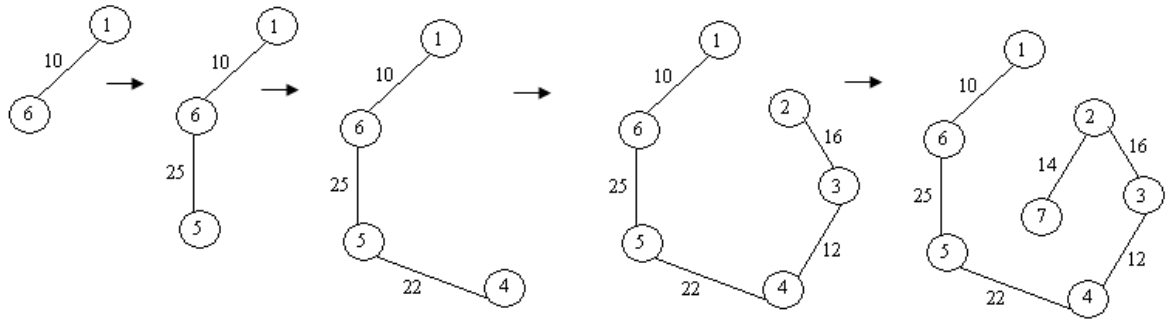
- Builds the tee edge by edge
- Next edge to be selected is one that result in a minimum increase in the sum of costs of the edges so far included
- Always verify that resultant is a tree

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;
    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

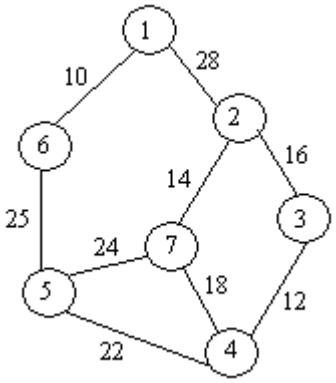
Kruskal's algorithm is another algorithm that finds a minimum spanning tree for a connected weighted graph. If the graph is not connected, then it finds a *minimum spanning forest* (a minimum spanning tree for each connected component).

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.

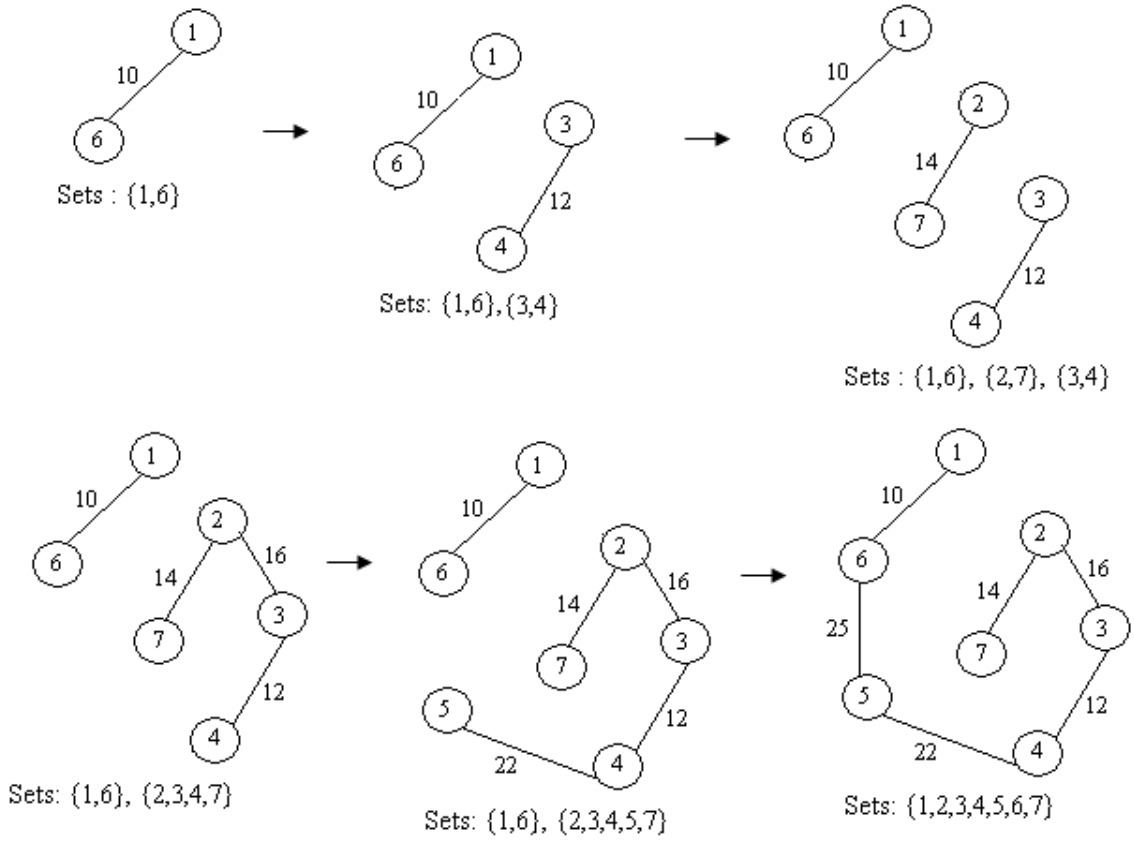
```
t = EMPTY;
while ((t has fewer than n-1 edges) && (E != EMPTY))
{
    choose an edge(v, w) from E of lowest cost;
    delete (v, w) from E;
    if (v, w) does not create a cycle in t
            add (v, w) to t;
    else
            discard (v, w);
}
```

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') $\cup$ {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 $O(n^2)$.

Theorem 3.3

Let *J* be a set of *k* jobs and $\sigma = i_1, i_2, \ldots , i_k$ such that $d_{i1}\ d_{i2} \ldots d_{ik}$. *J* is a feasible solution iff the jobs in *J* are processed in the order without violating any deadline.

Proof
If jobs in J can be processed without violating deadline, then J is feasible. If J is feasible then there exist an order σ. If J is feasible, there exist an order $\sigma' = r_1, ri_2, \dots, ri_k$ such that $d_{rq} \geq q$, $1 \leq q \leq k$. Assume that $\sigma' \neq \sigma$. Let a be the least index such that $r_a \neq i_a$. Let $r_b = i_a$ We know that b>a. In σ' we can interchange r_a and r_b. Since $d_{ra} \geq d_{rb}$, resultant $\sigma'' = s_1, si_2, \dots, si_k$ represent an order in which jobs can be processed without violating deadline. Continuing like this, σ' transforms to σ. Hence we have proved that J is a feasible solution.

Theorem 3.4
Greedy method always obtains an optimal solution for the Job sequencing problem.

Proof
Let (pi , d_i), be any instance of the problem. Let I and J be set of jobs selected by Greedy method and Optimal solution respectively. If I = J, then I is optimal. If $I \neq J$, and $J \subset I$, J cannot be optimal. Similarly, the possibility for $I \subset J$, ruled out by Greedy method. Hence there can be two elements and b such that $a \in I, a \notin J$ and $b \in J, b \notin I$. Let a be the highest profit element such that $a \in I, a \notin J$. Hence $p_a \geq p_b$ for all b in J not in I. Otherwise Greedy method would have selected b.

Now consider the schedule S_I and S_J. Let i be a job such that $i \in J$ and $i \in I$. Let i be scheduled from [t to t+1] in I and [t' to t'+1] in J. If t < t', interchange the jobs scheduled in [t' to t'+1] in S_I with i. If no job exist at [t' to t'+1] in I, move i to t'. If t'<t. so similar transformation in S_J. Doing this repeatedly, all the jobs common to I and J will be scheduled in same time. Let the new schedule be S_I' and S_J'. Consider interval [t_a to t_a+1] in S_I' in which job a is scheduled. Let job b be scheduled at that time in S_J'. From the choice of a, $p_a \geq p_b$. Scheduling a from [t_a to t_a+1] in S_J' and discarding b, we get J' = J – {b} U {a} which is also a feasible solution. Profit of J' is not less that profit of J. Doing this repeatedly, J transforms to I. Hence we can say that I is an optimal solution.

3.4 Optimal Storage on Tapes

Problem: n programs are to be stored on a computer tape of length l. Associated with each program t is a length l_i, $1 \leq i \leq n$. All programs can be stored on the tape iff the sum of length of programs is at most l. Whenever program is to be retrieved, tape is positioned at the front. If stored in the order $I = i_1, i_2, \dots, i_n$ time to retrieve i_j, given by t_j is proportional to $\sum_{k=1}^{j} l_{i_k}$. If all programs are equally often chosen, mean retrieval time $(MRT) = 1/n \sum_{j=1}^{n} t_j$. The problem is to find a permutation for n that minimizes MRT.

$$d[I] = 1/n \sum_{j=1}^{n} \sum_{k=1}^{j} l_{i_k}$$

Example 3.5
n = 3 $(l_1, l_2, l_3) = (5, 10, 3)$

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 +1 10 + 5 = 34

Greedy method chose the next program to be stored on tape as on that minimizes increase in d. Hence store in the non decreasing order of length.

Theorem 3.5

If $l_1 \le l_2 \le .. \le l_n$, then ordering $i_j = j$, $1 \le j \le n$ minimizes $\sum_{k=1}^{n} \sum_{j=1}^{k} l_{i_j}$ over all possible permutations of i_j.

Proof

Let $I = (i_1, i_2, \dots, i_n)$ be any permutation.

$$d(I) = \sum_{k=1}^{n} \sum_{j=1}^{k} l_{i_j} = \sum_{k=1}^{n} (n-k+1)\, l_{i_k}$$

There exist a and b such that a<b and $l_{ia} > l_{ib}$., Interchanging i_a and i_b we have

$$d(I') = \sum_{\substack{k \\ k \neq a \\ k \neq b}} (n-k+1)\, l_{ik} - (n-a+1) l_{ib} + (n-b+1) l_{ia}$$

$$d(I) - d(I') = [n-a+1]\,(l_{ib} - l_{ia}) + [n-b+1](l_{ia} - l_{ib})$$

$$d(I) - d(I') = (b-a)\,(l_{ia} - l_{ib}) > 0$$

No permutation which is not in decreasing order of l_i decreases the value of d. Also all permutation with decreasing order of l_i has same d. Hence we proved that ordering $i_j = j$, $1 \le j \le n$ minimizes the value of d.