# The Greedy Method

## General Method

- Most straightforward design technique
  - Most problems have $n$ inputs
  - Solution contains a subset of inputs that satisfies a given constraint
  - Feasible solution: Any subset that satisfies the constraint
  - Need to find a feasible solution that maximizes or minimizes a given *objective function* – optimal solution
- Used to determine a feasible solution that may or may not be optimal
  - At every point, make a decision that is locally optimal; and hope that it leads to a globally optimal solution
  - Leads to a powerful method for getting a solution that works well for a wide range of applications
    * The OPT algorithm for process scheduling, and its variant SRTN, in operating systems
  - May not guarantee the best solution
- Ultimate goal is to find a feasible solution that minimizes [or maximizes] an *objective function*; this solution is known as an *optimal solution*
- Devise an algorithm that works in stages (**subset paradigm**)
  - Consider the inputs in an order based on some selection procedure
    * Use some optimization measure for selection procedure
  - At every stage, examine an input to see whether it leads to an optimal solution
  - If the inclusion of input into partial solution yields an infeasible solution, discard the input; otherwise, add it to the partial solution

```
// Algorithm takes as input an array a of n elements

algorithm greedy ( a, n )
{
    solution = {};        // Initially empty
    for ( i = 0; i < n; i++ )
    {
        // Select an input from a and remove it from further consideration

        x = select ( a );

        if ( feasible ( solution, x ) )
            solution = solution + x;     // Union
    }

    return ( solution );
}
```

  - The algorithm `greedy` requires that the functions `select`, `feasible`, and `union` are properly implemented
- Ordering paradigm
  - Some algorithms do not need selection of an optimal subset but make decisions by looking at the inputs in some order
  - Each decision is made by using an optimization criterion that is computed using the decisions made so far

**Activity selection problem**

- Similar to process scheduling problem in operating systems
- Greedy algorithm efficiently computes an optimal solution
- Several competing activities require exclusive use of a common resource
- Goal is to select a set of maximum-size set of mutually compatible activities
  - Set $S$ of $n$ proposed activities, requiring exclusive use of a resource, such as a lecture hall

    $$S = \{a_1, a_2, \ldots, a_n\}$$

  - Each activity $a_i$ has a start time $s_i$ and a finish time $f_i$, such that $0 \leq s_i < f_i < \infty$
  - Activity $a_i$ takes place in the interval $[s_i, f_i)$, if selected
  - Activities $a_i$ and $a_j$ are compatible if intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap
    * Compatible if $s_i \geq f_j$ or $s_j \geq f_i$
  - Activity selection problem is to select a maximum-size subset of mutually compatible activities
  - Activities are assumed to be sorted in monotonically increasing order of finish time

    $$f_1 \leq f_2 \leq f_3 \leq \cdots \leq f_{n-1} \leq f_n$$

- Example
  - $S$ is given by

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $s_i$ | 1 | 3 | 0 | 5 | 3 | 5 | 6 | 8 | 8 | 2 | 12 |
| $f_i$ | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

  - Feasible solutions
    * $\{a_3, a_9, a_{11}\}$ – Mutually compatible activities but not maximum subset
    * $\{a_1, a_4, a_8, a_{11}\}$
    * $\{a_2, a_4, a_9, a_{11}\}$
- Optimal substructure of the activity-selection problem
  - Let $S_{ij}$ be the set of activities that start after $a_i$ finishes and before $a_j$ starts
  - Let $A_{ij}$ be the maximum set satisfying the constraints on $S_{ij}$
  - $A_{ij}$ includes some activity $a_k$
  - By including $a_k$ in optimal solution, we have to solve two subproblems: find mutually compatible activities in sets $S_{ik}$ and $S_{kj}$
  - Let $A_{ik} = A_{ij} \cap S_{ik}$ and $A_{kj} = A_{ij} \cap S_{kj}$
  - Then, $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$
  - $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$
  - The optimal solution $A_{ij}$ must also include optimal solutions to two subproblems $S_{ik}$ and $S_{kj}$

    $$|A_{ij}| = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max_{a_k \in S_{ij}} \{|A_{ik}| + |A_{kj}| + 1\} & \text{otherwise} \end{cases}$$

  - This leads to a dynamic programming solution but we can do better
- Making the greedy choice
  - Choose an activity that leaves the resources available for as many activities as possible

- One of the chosen activities must be the first one to finish
- Choose the activity $a_i \in S$ with the earliest $f_i$
- Since the activities are sorted in monotonically increasing order by $f_i$, the greedy choice is activity $a_1$
- The remaining subproblem is to find activities that start after $a_1$ finishes (compatible activities to $a_1$)
- Let $S_k = \{a_i \in S \,:\, s_i \geq f_k\}$
- Use the optimal substructure to solve this problem with the selection of $a_1$

  **Theorem 1** *Consider any nonempty subproblem $S_k$, and let $a_m \in S_k$ with earliest finish time. Then, $a_m$ is included in some maximum-size subset of mutually compatible activities of $S_k$.*

  **Proof** Let $A_k$ be a maximum-size subset of mutually compatible activities in $S_k$, and let $a_j \in A_k$ with the earliest finish time

  * If $a_j = a_m$, we are done since we have shown that $a_m$ is in some maximum-size subset of mutually compatible activities of $S_k$
  * If $a_j \neq a_m$, let the set $A'_k = A_k - \{a_j\} \cup \{a_m\}$ be $A_k$ but substituting $a_m$ for $a_j$
  * The activities in $A'_k$ are disjoint which follows since the activities in $A_k$ are disjoint, $a_j$ is the first activity in $A_k$ to finish, and $f_m \leq f_j$
  * Since $|A'_k| = |A_k|$, we conclude that $A'_k$ is a maximum-size subset of mutually compatible activities of $S_k$, and it includes $a_m$ QED

- Recursive greedy algorithm

```
// Return a maximum-size set of mutually compatible activities in S_k
// Assumption: n input activities are sorted by monotonically increasing
//             finish time

algorithm recursive_activity_selector (
    s,           // Input: Array containing start time of activities
    f,           // Input: Array containing finish time of activities
    k,           // Input: Index k to define the subproblem S_k to be solved
    n            // Input: Size of the original problem
    )
{
    m = k + 1;

    // Find the first activity in S_k to finish

    while ( m <= n and s[m] < f[k] )
        m = m + 1;

    if ( m <= n )
        return ( union ( {a_m}, recursive_activity_selector ( s, f, m, n ) ) );

    return ( NULL );
}
```

  - The implementation of the algorithm will add a fictitious activity $a_0$ with $f_0 = 0$, so that problem $S_0 \equiv S$
  - The initial call to solve the entire problem is

```
recursive_activity_selector ( s, f, 0, n );
```

- Iterative greedy algorithm

**Knapsack problem**

- Input: $n$ objects and a knapsack
- Each object $i$ has a weight $w_i$ and the knapsack has a capacity $m$
- A fraction of an object $x_i, 0 \leq x_i \leq 1$ yields a profit of $p_i \cdot x_i$
- Objective is to obtain a filling that maximizes the profit, under the weight constraint of $m$
- Formally,

$$\begin{array}{ll} \text{Maximize} & \sum_{i=1}^{n} p_i \cdot x_i \\ \text{subject to} & \sum_{i=1}^{n} w_i \cdot x_i \leq m \\ \text{and} & 0 \leq x_i \leq 1, 1 \leq i \leq n \\ \text{and} & \text{Each } p_i > 0 \text{ and } w_i > 0 \end{array}$$

- Problem instance: $n = 3$, $m = 20$, $P = (25, 24, 15)$, and $W = (18, 15, 10)$.
- Greedy strategy 1: Pick items with maximum profit per item.
  Solution: $(1, \frac{2}{15}, 0)$. Profit: 28.2
- Greedy strategy 2: Pick as many items as possible (smallest weight items first).
  Solution: $(0, \frac{2}{3}, 1)$. Profit: 31
- Greedy strategy 3: Pick items with maximum profit per unit weight.
  Solution: $(0, 1, \frac{1}{2})$. Profit: 31.5
- Items considered in the objective function: total profit, capacity used, and ratio of accumulated profit to capacity used

**Lemma 1** *In case $\sum_{i=1}^{n} w_i \leq m$, then, $x_i = 1, 1 \leq i \leq n$ is an optimal solution.*

**Lemma 2** *All optimal solutions will fit the knapsack exactly.*

- Algorithm

```
void greedy_knapsack ( m, n )
{
    // Solution vector is x[i], 0 <= i < n

    for ( i = 0; i < n; i++ )
        x[i] = 0.0;

    U = m;                               // Unused capacity
    for ( i = 0; ( i < n ) && ( w[i] <= U ); i++ )
    {
        x[i] = 1.0;
        U = U - w[i];
    }
    if ( i < n )
        x[i] = U / w[i];
}
```

**Theorem 2** *if $\frac{p_0}{w_0} \geq \frac{p_1}{w_1} \geq \cdots \frac{p_{n-1}}{w_{n-1}}$, then* `greedy_knapsack` *generates an optimal solution to the given instance of the knapsack problem.*

**Proof:** Let $x = (x_0, \ldots, x_{n-1})$ be the solution generated by `greedy_knapsack`

- If $\forall_i x_i = 1$, the solution is optimal

- Let $j$ be the least index such that $x_j \neq 1$. Then, the following holds:
  - $x_i = 1$ for $0 \leq i < j$
  - $x_i = 0$ for $j < i < n$
  - $0 \leq x_j < 1$
- Let $y = (y_0, \ldots, y_{n-1})$ be an optimal solution
  - From Lemma 2, we can assume that $\sum w_i y_i = m$
- Let $k$ be the least index such that $y_k \neq x_k$
  - Such a $k$ must exist since $x \neq y$
  - $y_k < x_k$; consider three possibilities
    1. $k < j$
       - $x_k = 1$
       - But $y_k \neq x_k \Rightarrow y_k < x_k$
    2. $k = j$
       - $\sum w_i x_i = m$, and $y_i = x_i$ for $0 \leq i < j$
       - Either $y_k < x_k$ or $\sum w_i y_i > m$
    3. $k > j$
       - $\sum w_i y_i > m$, which is not possible

**Tree vertex splitting**

- Directed and weighted binary tree
- Consider a network of power line transmission
- The transmission of power from one node to the other results in some loss, such as drop in voltage
- Each edge is labeled with the loss that occurs (edge weight)
- Network may not be able to tolerate losses beyond a certain level
- You can place boosters in the nodes to account for the losses

  **Definition 1** *Given a network and a loss tolerance level, the **tree vertex splitting problem** is to determine the optimal placement of boosters.*

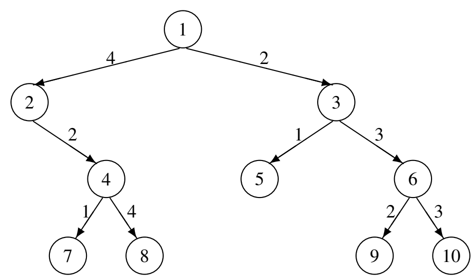
  You can place boosters only in the vertices and nowhere else

- More definitions
  - Let $T = (V, E, w)$ be a weighted directed tree
    - $V$ is the set of vertices
    - $E$ is the set of edges
    - $w$ is the weight function for the edges
    - $w_{ij}$ is the weight of the edge $\langle i, j \rangle \in E$
      We say that $w_{ij} = \infty$ if $\langle i, j \rangle \notin E$
    - A vertex with in-degree zero is called a *source vertex*
    - A vertex with out-degree zero is called a *sink vertex*
    - For any path $P \in T$, its *delay* $d(P)$ is defined to be the sum of the weights ($w_{ij}$) of that path, or

$$d(P) = \sum_{\langle i,j \rangle \in P} w_{ij}$$

  - $*$ Delay of the tree $T$, $d(T)$ is the maximum of all path delays
- Splitting vertices to create forest
  - $*$ Let $T/X$ be the forest that results when each vertex $u \in X$ is split into two nodes $u^i$ and $u^o$ such that all the edges $\langle u, j \rangle \in E$ [$\langle j, u \rangle \in E$] are replaced by edges of the form $\langle u^o, j \rangle \in E$ [$\langle j, u^i \rangle \in E$]
    - · Outbound edges from $u$ now leave from $u^o$
    - · Inbound edges to $u$ now enter at $u^i$
  - $*$ Split node is the booster station

- Tree vertex splitting problem is to identify a set $X \subseteq V$ of minimum cardinality (minimum number of booster stations) for which $d(T/X) \leq \delta$ for some specified tolerance limit $\delta$
  - TVSP has a solution only if the maximum edge weight is $\leq \delta$
- Given a weighted tree $T = (V, E, w)$ and a tolerance limit $\delta$, any $X \subseteq V$ is a feasible solution if $d(T/X) \leq \delta$
  - Given an $X$, we can compute $d(T/X)$ in $O(|V|)$ time
  - A trivial way of solving TVSP is to compute $d(T/X)$ for every $X \subseteq V$, leading to a possible $2^{|V|}$ computations




- Solve the above tree with $\delta = 5$
- Greedy solution for TVSP
  - We want to minimize the number of booster stations ($X$)
  - For each node $u \in V$, compute the maximum delay $d(u)$ from $u$ to any other node in its subtree
  - If $u$ has a parent $v$ such that $d(u) + w(v, u) > \delta$, split $u$ and set $d(u)$ to zero
  - Computation proceeds from leaves to root
  - Delay for each leaf node is zero
  - The delay for each node $v$ is computed from the delay for the set of its children $C(v)$

$$d(v) = \max_{u \in C(v)} \{d(u) + w(v, u)\}$$

    If $d(v) > \delta$, split $v$
  - The above algorithm computes the delay by visiting each node using post-order traversal

```
int tvs ( tree T, int delta )
{
    if ( T == NULL ) return ( 0 );      // Leaf node

    d_l = tvs ( T.left(), delta ); // Delay in left subtree
    d_r = tvs ( T.right(), delta ); // Delay in right subtree

    current_delay = max ( w_l + d_l,// Weight of left edge
                          w_r + d_r ); // Weight of right edge
```

```
        if ( current_delay > delta )
        {
            if ( w_l + d_l > delta )
            {
                write ( T.left().info() );
                d_l = 0;
            }
            if ( w_r + d_r > delta )
            {
                write ( T.right().info() );
                d_r = 0;
            }
        }
        current_delay = max ( w_l + d_l, w_r + d_r );
        return ( current_delay );
    }
```

- Algorithm `tvs` runs in $\Theta(n)$ time
  * `tvs` is called only once on each node in the tree
  * On each node, only a constant number of operations are performed, excluding the time for recursive calls

**Theorem 3** *Algorithm* `tvs` *outputs a minimum cardinality set* $U$ *such that* $d(T/U) \leq \delta$ *on any tree* $T$*, provided no edge of* $T$ *has weight* $> \delta$.

**Proof** by induction:

**Base case.** If the tree has only one node, the theorem is true.

**Induction hypothesis.** Assume that the theorem is true for all trees of size $\leq n$.

**Induction step.** Consider a tree $T$ of size $n + 1$

- Let $U$ be the set of nodes split by `tvs`
- Let $W$ be a minimum cardinality set such that $d(T/W) \leq \delta$
- We need to show that $|U| \leq |W|$
- If $|U| = 0$, the above is indeed true
- Otherwise
  * Let $x$ be the first vertex split by `tvs`
  * Let $T_x$ be the subtree rooted at $x$
  * Let $T' = T - T_x + x$ // Delete $T_x$ from $T$ except for $x$
  * $W$ has to have at least one node, $y$, from $T_x$
  * Let $W' = W - \{y\}$
  * If $\exists W*$ such that $|W*| < |W'|$ and $d\left(\frac{T'}{W*}\right) \leq \delta$, then since $d\left(\frac{T}{W*+\{x\}}\right) \leq \delta$, $W$ is not minimum cardinality split set for $T$
  * Thus, $W'$ has to be a minimum cardinality split set such that $d\left(\frac{T'}{W'}\right) \leq \delta$
- If `tvs` is run on tree $T'$, the set of split nodes output is $U - \{x\}$
- Since $T'$ has $\leq n$ nodes, $U - \{x\}$ is a minimum cardinality set split for $T'$
- This means that $|W'| \geq |U| - 1$, or $|W| \geq |U|$

**Elements of greedy strategy**

- Obtain a solution by making a series of choices

  - At every point, make the choice that seems best at that point
  - May not always result into optimal solution
- Steps for the greedy algorithm development
  1. Determine optimal substructure of problem
  2. Develop a recursive solution
  3. Show that if we develop a greedy choice, then only one subproblem remains
  4. Prove that it is always safe to make the greedy choice
  5. Develop a recursive algorithm to implement greedy strategy
  6. Convert the recursive algorithm to an iterative algorithm
- General design principles
  1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve
  2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe
  3. Demonstrate optimal substructure by showing that, having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice, we get an optimal solution to the original problem
- Greedy-choice property
  - Assemble a globally optimal solution by making locally optimal (greedy) choices
    * Make whatever choice seems best at the moment and then solve the remaining subproblem
    * Choice made by greedy algorithm may depend on the choices made so far but cannot depend on future choices or solutions to the subproblems
  - Must prove that a greedy choice at each step yields a globally optimal solution
    * Proof examines a globally optimal solution to some subproblem
    * It then shows how to modify the solution to substitute the greedy choice for some other choice, resulting in one similar, but smaller, subproblem
  - We may preprocess the input to make greedy choices quickly
- Optimal substructure
  - A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems
  - Assume that we arrived at a subproblem by making the greedy choice in the original problem
  - Argue that an optimal solution to the subproblem, combined with the greedy choice already made, yields an optimal solution to the original problem
  - The scheme implicitly uses induction on the subproblems to prove that making the greedy choice at every step produces an optimal solution

**Minimum Spanning Trees**

- Spanning tree (electronic circuit)

  **Definition 2** *Let $G = (V, E)$ be an undirected connected graph. A subgraph $t = (V, E')$ of $G$ is a **spanning tree** of $G$ iff $t$ is a tree.*

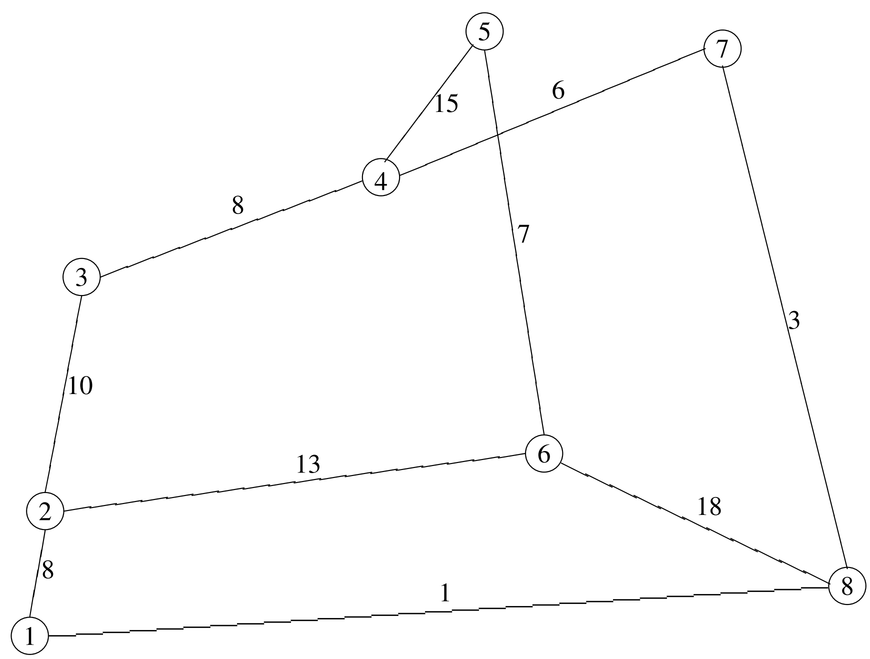
  - A spanning tree is a minimal subgraph $G'$ of a graph $G$ such that $V(G') = V(G)$ and $G'$ is connected

    * Let $B$ be the set of edges in $G$ that are not in the spanning tree
    * Adding an edge from $B$ to the spanning tree creates a cycle
  - Any connected graph with $n$ vertices must have $n-1$ edges
  - All connected graphs with $n-1$ edges are trees

- Minimum spanning tree
  - Spanning tree with minimum cost, based on edge weights

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

  - A graph and its minimum cost spanning tree




- Growing a minimum spanning tree
  - Manage a set $A$ that is always a subset of some minimum spanning tree
  - At each step, an edge $(u, v)$ is determined such that $A \cup (u, v)$ is also a subset of a minimum spanning tree
  - $(u, v)$ is called a *safe edge*

```
generic_MST (G,w)
{
    A = NULL;
    while A does not form a spanning tree
    {
        find an edge <u,v> that is safe for A
        Add <u,v> to A
    }
    return A;
```

- Prim's algorithm
  - Grow the tree one edge at a time
    - The edge is chosen based on some optimization criterion
    - Choose an edge that increases the overall weight of the tree by a minimum amount
    - Set of edges $A$ selected thus far forms a tree
    - The next edge $(u, v)$ to be included in $A$ is a minimum cost edge not in $A$ with the property that $A \cup \{(u, v)\}$ is also a tree
  - Start with any arbitrarily selected vertex
    - A way to select the starting point is to start with the two vertices of the minimum cost edge
    - The set $A$ of edges so far selected form a tree
    - The next edge $(u, v)$ to be included in $A$ is a minimum cost edge not in $A$ such that $A \cup \{(u, v)\}$ is also a tree
    - With each vertex, associate a value called `near[j]` with each vertex `j` that is not yet included in the tree
    - `near[j]` refers to the vertex in the tree such that `cost(j,near[j])` is minimum for all choices for `near[j]`
    - Change `near[j]` to $\infty$ for all vertices `j` that are already in the tree
  - Add a vertex to the set by looking for the closest vertex to the current tree
- **Kruskal's Algorithm**
  - A **cut** $(S, V - S)$ of an undirected graph $G = (V, E)$ is a partition of $V$
  - An edge $(u, v) \in E$ **crosses** the cut $(S, V - S)$ if one of its endpoints is in $S$ while the other is in $V - S$
  - Consider the edges of graph in non-decreasing order of cost
  - The set $t$ of edges selected at any point should be such that it should be possible to complete $t$ into a tree
    - $t$ is not required to be a tree at all stages but is not allowed to have a cycle either
  - Algorithm

```
MST_Kruskal ( G, t )
{
    // G is the graph, with edges E(G) and vertices V(G).
    // w(u,v) gives the weight of edge (u,v).
    // t is the set of edges in the minimum spanning tree.

    // Build a heap out of the edges using edge cost as the comparison criteria
    // using the heapify algorithm

    heap = heapify ( E(G) )
    t = NULL;

    // Change the parent of each vertex to a NULL
    // Each vertex is in different set

    for ( i = 0; i < |V(G)|; i++ )
        parent[i] = NULL

    i = 0

    while ( ( i < n - 1 ) && ! empty ( heap ) )
    {
        e = delete ( heap )   // Get minimum cost edge from heap
        adjust ( heap )       // Reheapify heap
```

```
            // Find both sides of edge e = (u,v) in the tree grown so far

            j = find ( u(e), t )
            k = find ( v(e), t )

            if ( j != k )
            {
                i++
                t[i,1] = u
                t[i,2] = v
                union ( j, k )
            }
        }
    }
```

**Single-source shortest paths**

- Travel from point $A$ to point $B$ on a map
- Questions:
  - Is there a path from $A$ to $B$?
  - If there is more than one path from $A$ to $B$, which is the shortest path?
- Graphs are considered to be digraphs to allow for one-way streets
- Starting point (vertex $v_0$) is called the *source* and the last vertex is called the *destination*
- Input: Directed graph $G = (V, E)$, a weighting function *cost* for all edges in $G$, and source vertex $v_0$
- Determine the shortest paths from $v_0$ to *all* the remaining vertices in $G$
- All weights are assumed to be positive
- Shortest path from $v_0$ to any vertex $v$ is an ordering of a subset of edges; hence the problem fits the ordering paradigm
- Greedy-based algorithm
  - Multi stage solution and optimization measure
  - Build the shortest paths one by one
  - Optimization based on the sum of lengths of all paths generated so far
    * Minimize the path length
    * If we have $i$ shortest paths, the next path to be constructed should be the next shortest minimum length path
    * In greedy method, generate these paths from $v_0$ to remaining vertices in non-decreasing order of path length
    * Start with a shortest path to nearest vertex from source