Greedy Algorithms

Rana Barua

Visiting Scientist, IAI, TCG CREST Kolkata

1 Greedy Algorithms

Typically, greedy algorithms apply to optimization problems in which we make a set of choices in order to arrive at an optimal solution. The idea of greedy algorithm is to make each choice in a locally optimal manner. For instance, if you want to pay an amount with minimum number of coins, at each step you would choose a coin with largest denomination available so as not to exceed the amount to be paid. We cannot always easily tell whether a greedy approach would be effective. A greedy algorithm makes a locally optimal choice in the hope that this would lead to a globally optimal solution. A greedy algorithm does not always yield an optimal solution, but in many cases they do. We first consider the *interval scheduling problem* which is also known as the scheduling problem.

1. **Scheduling**

   We have a collection of *jobs* or *tasks* to schedule on some machine Each job j has a given start time s_j and a given finish time f_j. Thus each job j is associated with an interval $[s_j, f_j]$. If two jobs overlap, we can't schedule them both.They are called **incompatible**. Our goal is to schedule as many jobs as possible on our machine.

   Example 1. Suppose the jobs are the following five intervals $[1, 3], [2, 4], [3, 5], [4, 6], [5, 7]$. Then best is to schedule the three jobs $[1, 3], [3, 5], [5, 7]$.

   Exercise 1. Call a job a *candidate job* if it has not yet been scheduled and is **compatible** with every already-scheduled job. Starting from the empty schedule, so long as one candidate job exists,
   - always add in the *shortest* candidate job;
   - always add in the candidate job with the *earliest start time*;
   - always add in the candidate job with the *fewest conflicts* with other candidate jobs.

   Give counter-examples to show that in each case we do not get an optimal solution.

   We shall now show that the **early finish time** greedy works and gives an optimal solution.
   Algorithm EFT

   Input: A list of intervals $\mathcal{I} = \{I_1, \dots, I_n\}$, where $I_j = [s_j, f_j]$ with $s_j < f_j$ for $1 \leq j \leq n$
   Output A maximun subset of pairwise compatible intervals in $\mathcal{I}$.

```
1.    Sort the input list $\mathcal{I}$ in non-decreasing order of finish time
2.    $f := 0$
3.    $S_0 = \phi$
4.      for $i \leftarrow 1$ to $n$ do
5.        if $f < s_i$ then
6.          $S_i = S_{i-1} \cup \{I_i\}$
7.          $f \leftarrow f_i$
8.        endif
9.      endfor
10      return $S_n$
```

Theorem 1. *Algorithm EFT gives an optimal solution in time $O(n \log n)$.*

Proof. We shall show by induction that, at every step,

- the solution produced by the algorithm so far can be extended to an optimal solution without removing any of the already-scheduled jobs.

The base step is clearly true. So assume it is true after i jobs have been scheduled. Let S ne the optimal solution that includes these i jobs sceduled by the algorithm so far. Let j be the job that the algorithm schedules next. If $j \in S$ then we are done. So assume that $j \notin S$. Let j' be the job in S with the earliest finish time that is not one of the i jobs scheduled by the algorithm. Clearly, j' is a candidate job since it is not in conflict with any other job in S and S includes the i jobs scheduled by the algorithm so far. Then it must be the case that the finish time of j' is greater than equal to the finish time of j (Why?) Thus j is not in conflict with any of the jobs in S that starts after j' finishes. Also, j is not in conflict with any job in S that finishes earlier that the start time of j', since it must be one of the i jobs scheduled by the algorithm. This means that $S^* = (S - \{j'\}) \cup \{j\}$ is an new optimal solution that includes all the i jobs together with j. So the invariant is maintained.
Thus the invariant is maintained throughout the algorithm so when it halts, the algorithm must have found an optimal solution. This completes the proof. □

Remark 1. Given a schedule $\mathcal{I} = \{I_1, \ldots, I_n\}$ construct a graph $G = (V, E)$ as follows. For every interval I_i create a vertex v_i and two vertices are adjacent i.e. $\{v_i, v_j\} \in E$ iff their corresponding intervals overlap. Then computing a maximum subset of pairwise compatible intervals is equivalent to computing a maximum independent set in G.

2. **Minimum Spanning Trees**

We shall consider two greedy algorithms for finding a *minimum spanning tree* of a given weighted connected graph. The first one is due to Priml and the other due to Krushkal.

Definition 1. *Let $G = (V, E)$ be a connected graph. A subgraph (V, T) is called a* **spanning tree** *if T is a tree. Let $w(,)$ be a real-valued function on E. T is said to be a minimum spanning tree if its weight*

$$w(T) = \sum_{e \in E} w(e)$$

is as small as possible.

We shall construct the minimum spanning tree by successively selecting edges to included in the tree. We shall ensure that after the inclusion of the new edge, the selected edges X form a subset of some minimum spanning tree. This is guaranteed by the following **cut property**.

Lemma 1. *Let $X \subseteq T$. where T is an MST for a graph $G = (V, E)$. Let $S \subseteq V$ such that no edge in X crosses between S and $V - s$ i.e. there is no edge in E with one end point in S and the other end point in $V - S$. Among the edges crossing between S and $V - S$, let e be the edge with minimum weight. Then $X \cup \{e\} \subseteq T'$ for some MST T'.*

Proof. If $e \in T$ then we are done. So assume that $e \notin T$. Adding e to T creates a unique cycle. Since e crosses between S and $V - S$, as we traverse the cycle, we must cross back along some other edge to come to the starting point. Thus there is an edge $e' \neq e$ in the cycle that crosses between S and $V - S$. Let $T' = T \cup \{e\} - \{e'\}$. Clearly, T' is a spanning tree (why?) By our choice of e we have $w(e') \geq w(e)$. Hence $w(T') \leq w(T)$. Since T is a minimum spanning tree, T' is also an MST and $w(e') = w(e)$. Since X has no edge crossing between S and $V - S$, $X \subseteq T'$ and hence $X \cup \{e\} \subseteq T'$. This completes the proof. □

Prim's Algorithm

In the case of Prim's algorithm, X consists of a single tree and S is the set of vertices of the tree. It grows into single tree adding a vertex at each step until it becomes a minimum spanning tree. The tree starts from a single arbitrary root. At each step a light edge connecting a vertex in S to a vertex in $V - S$ is added to the tree.

The key to implementing Prim's algorithm efficiently is to make it easy to select a new edge to be added to the tree formed by the edges in X. In the pseudocode below, the connected graph G and the root r of the minimum spanning tree to be grown are inputs to the algorithm. During execution of the algorithm, all vertices that are not in the tree reside in a priority queue Q based on a key field. For each vertex $v, key[v]$ is the minimum weight of any edge connecting v to a vertex in the tree; by convention, $key[v] = \infty$ if there is no such edge. The field $\pi[v]$ names the "parent" of v in the tree. During the algorithm, the set X is kept implicitly as

$$X = \{(v, \pi[v]) : v \in V - r - Q\}.$$

When the algorithm terminates, the priority queue Q becomes empty. The spanning tree is thus

$$X = \{(v, \pi[v]) : v \in V - \{r\}\}.$$

Algorithm Prim(G, w, r)
Input: A weighted connected graph $G = (E, V, w)$, where w is a weight function
Output: A minimum spanning tree T

```
1.   $Q \leftarrow V$
2.   for each $u \in Q$
3.       do $key[u] := \infty$
4.   $key[r] := 0$
5.   $\pi[r] \leftarrow NIL$
6.   while $Q \neq \phi$
7.       do $u \leftarrow EXTRACT - MIN(Q)$
8.           for each $v \in Adj[u]$
9.               do if $v \in Q$ and $w(u, v) < key[v]$
10.                  then $\pi[v] \leftarrow u$
11.                       $key[v] \leftarrow w(u, v)$
```

Before the execution of the **while** loop, we have
- $X = \{)v, \pi(v)) : v \in V - \{r\} - Q$
- $V - Q$ is the set of vertices already placed in the minimum spanning tree.
- For every $v \in Q$, if $\pi[v] \neq NIL$, then $k[v]$ is the weight of the light edge $(v, k[v])$ connecting v

with a vertex already placed in the tree.
Line 7 identifies a vertex $u \in Q$ that is incident with a light edge that crosses the cut $V - Q$ and Q. Removing u adds it to the set $V - Q$ of vertices of the tree, thus adding the edge $(u, \pi[u])$ to X.
Correctness and Implementation:
The correctness of the algorithm follows from Lemma 1.
The running time of Prim's algorithm depends on how we implement the priority queue Q. If Q is implemented as a binary heap, then building a heap takes $O(n)$ time. The loop is executed n times, and since each EXTRACT-MIN takes $O(\log n)$ times, the total time for the execution of the loop is $O(n \log n)$ time. The **for** loop is executed $O(e)$ times, and since line 11 involves updating a key, it can be implemented in $O(\log n)$ time. Thus the total time for Prim's algorithm is $O(e \log n + n \log n) = O(e \log n)$.

Krushkal's Algorithm:

Krushkal's algorithm adopts a different strategy. Instead of growing a single tree, Krushkal's algorithm adds the lighest possible edge to the tree at each step. It starts with the edges sorted by increasing order of weight. Initially $X = \phi$ and each vertex of the graph is regarded as a trivial tree. Each edge in the sorted list is examined in order and if the end points are in the same tree then it is discarded. Otherwise, it is included in the set X and this causes the two trees containing the end points of this edge to merge into a single tree. Observe that by this process we implicitly choose a set $S \subseteq V$ with no edge in X crossing between S and $V - S$
To implement Krushkal's algorithm, given a forest of trees, we must decide whether two given vertices are in the same tree. For the purpose of implementation, each tree is represented as a set of vertices of that tree. We also need to update our data structure to reflect the merging of two trees into a single tree. Thus our data structure will maintain a collection of disjoint sets and support the following three operations.

- MAKESET(x): Create a new set x consisting of a single element x.
- FIND(x): Given an element x, it finds the set containing x.
- UNION(x,y): Replace the set containing x and the set containing y by their union.

```
**Krushkal's Algorithm**$(G = (V, E))$
      $X \leftarrow \phi$
      Sort $E$ by weight
        **for** $u \in V$
           $MAKESET(u)$
        **end for**
        **for**$(u, v) \in E$ in increasing order **do**
           **if** $FIND(u) \neq FIND(v)$ **then**
              $X \leftarrow X \cup \{(u, v)\}$
              $UNION(u, v)$
        **end for**
        **return** $X$
```

Correctness and Running Time:
The correctness of the algorithm follows from Lemma 1.
The running time of the algorithm, assuming that the edges are given in sorted order, is dominated by the set operations FIND and UNION. There are $n-1$ UNION operations and $2e$ FIND operations, and these can be implemented by a fast UNION-FIND that takes $O(e \log n)$ time. If

the edges need to be sorted, then it will require $O(e \log e)$ time and that is the dominant part of the running time. □

Exercise 2. (a) Let $G = (V, E)$ be a weighted connected graph. Show that if the weights are distinct, then there is a unique minimum spanning tree.

(b) Prove the following **exchange property** satisfied by spanning trees.

Let T and T' be spanning trees in $G = (V, E)$. Given an edge $e' \in T' - T$, there exists an edge $e \in T - T'$ such that $(T - \{e\}) \cup \{e'\}$ is also a spanning tree.

Use this exchange property to show that one can "walk" from any spanning tree T to a minimum spanning tree $\hat{T}$.

3. Clustering

Suppose we are given a set of objects *e.g.* texts, photos, organisms etc and our aim is to organize or classify them into coherent groups. We shall assume that there is a *distance function* on the objects with the understanding that objects that are at a larger distance are less similar to each other.
Thus, given a distance function on the objects, the clustering problem seeks to divide them in groups so that objects within the groups are "near" and the objects in diferent groups are "far apart".

Clusterings of Maximum Spacing: Suppose we are given a set U of n objects labeled $p_1, p_2, \ldots, p_n$. Suppose we are seeking to divide the objects in U into k groups, for a given parameter k. A partition of U into k non-empty sets $C_1, C_2, \ldots, C_k$ is called a $k-$*clustering* of U. We define the *spacing* of a k-clustering to be the minimum distance of any pair of points lying in different clusters. Our goal is to find a k-clustering with the maximum possible spacing.

The Algorithm.
To find a clustering of maximum spacing, we consider growing a graph on the vertex set U. The connected components will be the clusters. Thus we start by drawing an edge between the closest pair of points. Then we draw an edge between the next closest pair of points. We keep adding edges between pair of points, in order of increasing distance $d(p_i, p_j)$. In this way, we grow a graph H on U edge by edge, with the connected components corresponding to the clusters. Observe that our graph growing process never creates a cycle, so H is actually a union of trees. Each time we add a new edge that crosses betwen two trees, we merge the corresponding clusters. This is known as *single-link clustering.*
Note that our procedure is exactly Krushkal's MST Algorithm. The only difference is that we stop when we obtain the k connected components. Clearly, the k components are obtained by running Krushkal's algorithm for $n - k$ steps. In other words, we are running Krushkal's algorithm but stopping just before adding its last $k - 1$ edges. This is equivalent to running the full Krushkal's algorithm and then deleting the $k - 1$ most expensive edges. We then define the k-clustering to be the resulting connected components $C_1, C_2, \ldots, C_k$.

Correctness and Complexity.
The complexity of the algorithm is exactly that of Krushkal's algorithm.
To see that the algorithm correctly gives us k-clustering with maximum spacing, we prove the following.

Theorem 2. *The components $C_1, \dots, C_k$ obtained by deleting the $k-1$ most expensive edges from the minimum spanning tree T form a k-clustering of maximum spacing.*

Proof. Let $\mathcal{C}$ denote the clustering $C_1, \dots, C_k$. The spacing of $\mathcal{C}$ is precisely the $k-1$st most expensive edge in the minimum spanning tree *i.e.* the length of the edge that Krushkal's algorithm would have added next, at the moment we stopped it. Denote it by d^*. Let $\mathcal{C}'$ be any other k-clustering that partitions U into k non-empty sets $C'_1, \dots, C'_k$. We must show that the spacing of $\mathcal{C}'$ is at most d^*.
If every cluster $C_i, 1 \leq i \leq k$ of $\mathcal{C}$ is a subset of some cluster of $\mathcal{C}'$, then $\mathcal{C} = \mathcal{C}'$. (Why?) Since $\mathcal{C} \neq \mathcal{C}'$, there must be a cluster C_r in $\mathcal{C}$ which is not a subset of any cluster in $\mathcal{C}'$. Hence there exists two points p_i, p_j in C_r which belong to two different clusters in $\mathcal{C}'$. Let $p_i \in C'_s$ and $p_j \in C'_t \neq C'_s$.
Let P be the path in C_r from p_i to p_j. Krushkal's algorithm must have added all the edges of P before we stopped it. This means that the length of each edge of P is at most d^*. Now, observe that $p_i \in C'_s$ and $p_j \notin C'_s$. Let p' be the first node on P that is not in C'_s and let p be the node on P that comes just before p'. As argued above, $d(p,p') \leq d^*$, since the edge was added by Krushkal's algorithm. But p and p' belong to different clusters of the clustering $\mathcal{C}'$. Thus the spacing of $\mathcal{C}'$ is at most $d(p,p') \leq d^*$. This completes the proof □

4. Huffman Codes and Data Compression

Since computers operate on bit strings we need an encoding scheme that takes texts written in a richer alphabet, say the English alphabet, and convert this text into a long string of bits. For instance, we may represent each letter of the English alphabet by a 5-bit string. In such a case, to encode a text consisting of 100 letters we would require 500 bits. If however, the letters a and e occur in that text 45 and 30 times respectively, then if we encode a by 0 and e by 1 then we would be requiring only 200 bits to encode the text. Thus we need a variable length encoding scheme where frequently occuring strings are encoded with shorter strings and less frequent letters are encoded with longer strings.
Prefix Codes: If we have an encoding scheme where that there is a pair of letters such that the encoding of one is a prefix of the encoding of the other, then there would be an ambiguity while decoding. Thus one prefers a *prefix code.*

Definition 2. *A prefix code on a set of letters S is a function $\gamma : S \to \{0,1\}^*$ such that for any two distinct letters $x, y \in S$, neither $\gamma(x)$ nor $\gamma(y)$ is a prefix of the other.*

Thus if we have a text consisting of the sequence of letters $x_1, \dots, x_n$, then we can simply encode it as the bit string $\gamma(x_1) \dots . \gamma(x_n)$. The original text can easily be recovered according to the following rule.

i. Scan the bit string from left to right
ii If the first few bits is an encoding of a letter, then it must be unique due to the prefix property. Recover the letter and remove the codeword from the string.
iii Go to step i.

Optimal Prefix Code: Our aim is to have a prefix code γ with more frequently occuring letters having shorter encodings. Suppose, for each letter $x \in S$, we have the frequency f_x representing the fraction of the letters in the text that are equal to x. Thus if the text consists of n letters, then nf_x of these are equal to x. So the length of the encoding is $\sum_{x \in S} nf_x|\gamma(x)|$. So the average number of bits required per letter is

$$ABL(\gamma) = \sum_{x \in S} f_x|\gamma(x)|.$$

Given an alphabet S and a set of frquencies of the letters in the alphabet, we would like to have a prefix code such that the average number of bits required per letter is minimum. Such a code is called an **optimal prefix code.**
Prefix Code using Binary Trees: Given an alphabet S, consider a binary tree where the number of leaves is equal to the number of letters. Label each leaf with a letter. We get an encoding of each letter as follows. Consider a leaf with label x. To get an encoding of x, we consider the path from the root to the leaf. Each time you traverse left write down a 0 and each time you traverse right write 1. Then the resulting bit string is an encoding of the letter x. It is easy to check that the resulting code is a prefix code.

Conversely, given a prefix code we can recursively construct a binary tree as follows. All letters $x \in S$ whose encodings start with a 0 will be labels of the left subtree and all the letters $y \in S$ whose encodings start with a 1 will be labels of the right subtree. We construct these two subtrees recursively using this rule.

Thus the length of the encoding of a letter $x \in S$ is simply the length of the path from the root to the leaf with label x. We call this the **depth** of the leaf and we denote the depth of a leaf node v in T by $depth_T(v)$.

Call a binary tree **full**, if each node that is not a leaf node has two children. We now prove

Lemma 2. *The binary tree corresponding to the optimal prefix code if full.*

Proof. Let T be a binary tree corresponding to the optimal binary tree. Let u be a node of T with exactly one child v. If u was a root node then delete u and make v the root. Otherwise, let w be the parent of u. Delete u and make v a child of w. Let T' be the resulting tree. By this change the number of bits needed to encode any leaf of the subtree at u decreases. The encodings of other leaf nodes remain unchanged. Hence the average number of bits per letter decreases, contradicting the optimality of T. $\square$

Lemma 3. *Let T^* be the binary tree corresponding to an optimal prefix code. Let u and v be leaf nodes of T^*. Suppose $depth(u) < depth(v)$. Let the labels of u, v be y and z respectively, $y, z \in S$. Then $f_y \geq f_z$.*

Proof. Let T be the tree obtained by interchanging the labels of u and v. Then $ABL(T) - ABL(T^*) = f_y depth(v) - f_y depth(u) + f_z depth(u) - f_z depth(v) = (f_y - f_z)(depth(v) - depth(u)) \geq 0$, by the optimality of T^*. Since depth(u)-depth(v) > 0, we have $f_y - f_z \geq 0$ i.e. $f_y \geq f_z$. $\square$
This lemma shows that the letter with the least frequency will have the maximum depth. This gives us the following

Lemma 4. *There is an optimal prefix code with corresponding tree T^*, in which the two lowest frequency letters are assigned to leaves that are siblings in T^*.*

Proof. Let v be a leaf in T^* whose depth is as large as possible. Let u be its parent in T^*. By Lemma 2, T^* is full and hence, u has another child, say w. Clearly, v and w are siblings. Observe that w must be a leaf node. If not, then w would have a child whose depth would be larger than v, contradicting our choice of v. Thus v and w are leaves in T^* and are siblings. $\square$

Algorithm for Optimal Prefix Code
Huffman's Algorithm

Input: An alphabet S with given frquencies $\{f_x; x \in S\}$
Output: A binary tree T representing an optimal prefix code.

```
  If $S$ contaings two letters then
    encode one letter with 0 and the other letter with 1
Else
    Let $y^*$ and $z^*$ be the letters with the lowest frquency
    Form a new alphabet $S'$ by deleting the letters $y^*, z^*$ and
        replacing them with a new letter $w$ of frquency $f_{y^*} + f_{z^*}$.
    Recursively construct a prefix code $\gamma'$ for $S'$ with tree $T'$
     Define a prefix code $\gamma$ for $S$ as follows:
        Start with $T'$
        Take the leaf node labeled $w$ and add two children below it
        Label them $y^*$ and $z^*$
Endif
```

Theorem 3. *The Huffman code for a given alphabet achieves the minimum average number of bits per letter of any prefix code.*

Proof. We shall prove this by induction on the size of the alphabet. If the alphabet contains two letters then clearly, the code is optimal. So assume that the algorithm yields optimal prefix code for every alphabet of size $k - 1$ and let S be an alphabet of size k.
Observe that the algorithm merges two lowest frequency letters $y^*, z^* \in S$ into a single letter w to obtain an alphabet of smaller size. By induction hypothesis, the algorithm produces an optimal prefix code represented by the tree T'. We now claim that

$$ABL(T') = ABL(T) - f_w. \tag{1}$$

First observe that the depth of each letter other than y^*, z^* are the same in both T and T'. Also the depth of each of y^*, z^* is one more than the depth of w in T'. Hence

$$\begin{aligned}
ABL(T) &= \sum_{x \in S} f_x depth_T(x) \\
&= f_{y^*} depth_T(y^*) + f_{z^*} depth_T(z^*) + \sum_{x \neq y^*, z^*} f_x depth_{T'}(x) \\
&= (f_{y^*} + f_{z^*})(1 + depth_{T'}(w)) + \sum_{x \neq y^*, z^*} f_x depth_{T'}(x) \\
&= f_w(1 + depth_{T'}(w) + \sum_{x \neq y^*, z^*} f_x depth_{T'}(x) \\
&= f_w + f_w depth_{T'}(w) + \sum_{x \neq y^*, z^*} f_x depth_{T'}(x) \\
&= f_w + \sum_{x \in S'} f_x depth_{T'}(x) \\
&= f_w + ABL(T').
\end{aligned}$$

Hence the claim.

Now suppose that T obtained by Huffman's algorithm is not optimal. Then there is a labeled binary tree Z such that $ABL(Z) < ABL(T)$. By Lemma 4, the leaves representing y^* and z^* are siblings in Z. If we delete the leaves labeled y^* and z^* from Z and label their parent with w, we obtain a prefix code for S'. Note that Z' is obtained from Z in the same way T' was obtained from T. Hence, by (1), we have $ABL(Z') = ABL(Z) - f_w$. By our assumption

$$ABL(Z) < ABL(T).$$

Therefore, $ABL(Z) - f_w < ABL(T) - f_w$ i.e. $ABL(Z') < ABL(T')$. This contradicts the optimality of T' and the proof is complete. □

Exercise 3. Show that the complexity of Huffman's algorithm is $O(n^2)$.
Show that by usibng priority queue it can be improved to $On \log n)$.

5. **Fractional Knapsack**

Given K and a set of n items with weights $w_1, \ldots, w_n$ and value $v_1, \ldots, v_n$ respectively, find $0 \le x_i \le 1, i = 1, 2, \ldots, n$ such that

$$\sum_{i=1}^{n} x_i w_i \le K$$

and the following is maximized

$$\sum_{i=1}^{n} x_i v_i.$$

Greedy Algorithm for Fractional Knapsack

- Calculate the value-per-unit $\rho_i = v_i/w_i, i = 1, 2, \ldots, n$
- Sort the items by decreasing ρ_i.
  Let the sorted item sequence be $1, 2, \ldots, n$ and for each i
  the corresponding value-per-unit and weight be ρ_i and w_i respectively.
- Let k be the current weight limit. Initially $k = K$.
  In each iteration we chose item i from the head of the unselected list.
  - If $k \ge w_i$ set $x_i := 1$(item i is chosen) and set $k \leftarrow k - w_i$.
  - If $k < w_i$ set $x_i := k/w_i$ (we take a fraction k/w_i of item i) and halt.

Theorem 4. *The greedy algorithm gives an optimal solution in time $O(n \log n)$.*

Proof. Given a set of n item $\{1, 2, \ldots, n\}$, assume items sorted by per-unit values:

$$\rho_1 \ge \rho_2 \ge \ldots \ge \rho_n.$$

Let the greedy solution be $X = (x_1, x_2, \ldots, x_n)$.
 •x_i denote the fraction of item i taken.
Consider any optimal solution $O = (y_1, y_2, \ldots, y_n)$.
 •y_i is the fraction of item i in O.
 • The knapsack must be full in both X and O.

$$\sum_{i=1}^{n} x_i w_i = \sum_{i=1}^{n} y_i w_i = K.$$

If $X = O$, then X is an optimal solution. So assume that $X \neq O$. Let i be the smallest index such that $x_i \neq y_i$. Since our greedy algorithm always take as much as it can, $x_i > y_i$. Let $x = x_i - y_i$. Consider the following new solution O' constructed from O.
For $j < i$, set $y'_j = y_j = x_j$. Set $y'_i = x_i$
In O, from items $i+1$ to n remove weights $\epsilon_{i_1}, \ldots, \epsilon_n$ such that the total weight removed equals xw_i. This results in decrease in value by $\epsilon_{i+1}\rho_{i+1} + \ldots + \epsilon_n\rho_n \leq \epsilon_{i+1}\rho_i + \ldots + \epsilon_n\rho_i = xw_i\rho_i = xv_i$, the value added to item i in O'. Hence, the total value in solution O' is greater or equal to the total value of solution O. Since, O is an optimal solution, the two values are the same. Thus O' is also optimal. Repeating this process, we will eventually convert O to X without changing the total value. Thus X is also optimal. $\square$

6. **Single Source Shortest Paths**

Definition 3. *Let $G = (E, V)$ be a weighted directed graph with weight function $w : E \to \mathbb{R}$. The weight of a path in G is the sum of the weights of the edges of the path.*
The single source shortest path problem is to find, for each $v \in V$, the smallest weight path from the source s to v i.e. the shortest path from s to v. We define

$$\delta(s,v) = \begin{cases} the\ shortest\ path\ from\ s\ to\ v & if \quad v\ is\ reachable\ from\ s \\ \infty & if \quad there\ is\ no\ path\ from\ s\ to\ v \end{cases}.$$

Dijkstra's Algorithm

We assume here that weight on each edge is non-negative. Dijkstra's algorithm constructs a set S of vertices whose shortest distances from the source is known. At each step, we add to S that remaining vertex v whose distance from the source is the shortest of the remaining vertices. If all edges have non-negative weights, then one can show that the path from the source to v passes only through vertices in S. Thus it is only necessary to record, for each vertex v, the shortest distance from the source to v along a path that passes only through vertices in S.

Dijkstra's Algorithm
Input: A directed graph $G = (V, E)$, a source $s \in V$ and weight function $w : E \to \mathbb{R}^+$ from the edges to the non-negative reals. We take $w(v_i, v_j) = +\infty$ if (v_i, v_j) is not an edge, $v_i \neq v_j$ and $w(v, v) = 0$.
Output: For each $v \in V$, the minimum over all paths p from s to v of the sum of the weights of the edges of p.
Method: We construct a set $S \subseteq V$ such that the shortest path from the source to each vertex v in S lies wholly in S. The array $D[v]$ contains the weight of the current shortest path from s to v passing only through vertices of S.

```
        begin
1.          $S \leftarrow \{s\}$
2.          $D[s] \leftarrow 0$
3.          for each $v$ in $V - \{s\}$ do $D[v] \leftarrow w(s,v)$
4.          while $S \neq V$
              begin
5.                choose a vertex $z \in V - S$ such that $D[z]$ is minimum.
6.                add $z$ to $S$
7.                for each $v$ in $V - S$ do
8.                    $D[v] \leftarrow Min\{D[v], D[z] + w(z,v)\}$
              end
        end
```

Theorem 5. *Dijkstra's algorithm computes the weight of the shortest path in $O(n^2)$ time.*

Proof. Lines 1–3 clearly require $O(n)$ time. The **for** loop of lines 7-8 requires $O(n)$ steps as does the selection of z at line 5. Since the **while** loop is executed in $O(n)$ time, the total cost is $O(n^2)$.
Correctness. By induction on $|S|$ we prove that for each $v \in S$, $D[v]$ is equal to the length of the shortest path from s to v. Moreover, for all $v \in V - S$, $D[v]$ is the length of the shortest path from s to v that lies wholly within S, except for v itself.
Basis: $|S| = 1$. The shortest path from s to itself has length 0, and a path from s to v, wholly within S except for v, consists of a single edge (s, v). Thus lines 2 and 3 correctly initialize the D array.
Inductive step: Suppose vertex z is selected at line 5. If $D[z]$ is not the length of the shortest path, then there must be a path p from s to z that contains a vertex, other than z, outside S. Let v be the first such vertex. The distance of s to v is shorter than $D[z]$. Moreover, the shortest path from s to v lies wholly within S except for v. By induction hypothesis $D[v]$, is the length of the shortest path from s to v. Hence

$$D[v] \leq length\ of\ p < D[z],$$

contrary to our choice of z. Such a path p does not exist and $D[z]$ is the length of the shortest path from s to z.
The second part of the induction hypothesis holds in view of line 8. □