Greedy Algorithms

CMSC 27300: Honors Theory of Algorithms

January 9-13, 2023

Corresponding section(s) of Kleinberg-Tardos: 4.1, 4.2, 4.4, 4.5

1 Overview

Greedy algorithms maximize the immediate payoff according to some rule (which depends on the problem and the algorithm). Greedy algorithms are not always optimal, as it may be better to take a smaller gain now in order to obtain a larger gain later. That said, greedy algorithms have the advantage that they are relatively simple and there are many problems where there is a greedy algorithm which is optimal or at least gives a good approximation.

2 Variants of Interval Scheduling

Problem 2.1 (Interval Scheduling: Maximizing the Number of Jobs Accepted)**.** *In this interval scheduling problem, we have one processor and we are given n jobs. Each job has a set time interval $[a_i, b_i]$ when it must use the processor if it is accepted. We are then asked to find a schedule S which accepts the maximum number of jobs without having two jobs which run on the processor at the same time.*

This problem can be stated more precisely using the following definition:

Definition 2.2. *Given n intervals $[a_1, b_1], \ldots, [a_n, b_n]$, we say that a sequence $S = (i_1, \ldots, i_m)$ of elements of $[n]$ is a valid schedule if $\forall j \in [m-1]$ $\left(b_{i_j} \leq a_{i_{j+1}}\right)$ (the jobs $i_1, \ldots, i_m$ are in sequential order and job i_{j+1} does not start before job i_j finishes). We define $|S|$ to be the number of elements in S (i.e. the number of jobs accepted).*

Precise problem statement: Given n intervals $[a_1, b_1], \ldots, [a_n, b_n]$, find a valid schedule $S = (i_1, \ldots, i_m)$ which maximizes $|S| = m$.

Algorithm 2.3. *This problem can be solved with the following greedy algorithm: When choosing the next job, always choose the job which can be finished first. More precisely,*

> *Stored data: We keep track of a valid schedule $S_k = (i_1, \ldots, i_k)$ of the jobs which we have accepted so far and the time $t_k = b_{i_k}$ when all of these jobs will be complete.*
>
> *Initialization: We start with $S_0 = \emptyset$ and $t_0 = 0$.*

Iterative step: Given $S_k = (i_1, \ldots, i_k)$ and t_k we choose the next job i_{k+1} from the set $\{i \in [n] : a_i \geq t_k = b_{i_k}\}$ of jobs which are still available so that $b_{i_{k+1}}$ is minimized. We then take $S_{k+1} = (i_1, \ldots, i_k, i_{k+1})$ and $t_{k+1} = b_{i_{k+1}}$. If there are no available jobs left then the algorithm terminates and we take $S = S_k$.

Theorem 2.4. *If the greedy algorithm gives a valid schedule $S = (i_1, \ldots, i_m)$ then for any other valid schedule $S' = (i'_1, \ldots, i'_{m'})$, $m' \leq m$*

Proof. The intuition is that by always choosing the job which finishes first, S completes jobs as fast as possible and thus it is impossible to get ahead of S. We can turn this intuition into a proof using the following lemma, which says that the greedy algorithm always stays ahead:

Definition 2.5. *Given a valid schedule $S = (i_1, \ldots, i_m)$, for all $j \in [m]$, define $t_j(S) = b_{i_j}$ to be the time at which the jth job in S finishes. For all $j > m$, define $t_j(S) = \infty$*

Lemma 2.6. *If $S = (i_1, \ldots, i_m)$ is the schedule given by the greedy algorithm then for any other valid schedule $S' = (i'_1, \ldots, i'_{m'})$, for all $j \in \mathbb{N}$, either $t_j(S) \leq t_j(S')$ or $t_j(S') = \infty$.*

Proof. We prove this lemma by induction. The base case $j = 1$ is trivial as $t_1(S) = b_{i_1} \leq b_{i'_1} = t_1(S')$. For the inductive step, assume that $t_k(S) \leq t_k(S')$. If $t_{k+1}(S') = \infty$ then we are done. Otherwise, note that $a_{i'_{k+1}} \geq t_k(S') \geq t_k(S)$ so job i'_{k+1} is available to S. Since S always chooses the job with the earliest completion time out of the available jobs, $t_{k+1}(S) = b_{i_{k+1}} \leq b_{i'_{k+1}} = t_{k+1}(S')$, as needed. □

Applying this lemma with $j = m + 1$, $t_{m+1}(S) = \infty$ so we must have that $t_{m+1}(S') = \infty$. Thus, $m' \leq m$, as needed. □

Remark 2.7. *This algorithm can be implemented in $O(nlogn)$ time by first sorting the jobs in increasing order of b_i and then going through these jobs one by one and accepting any job which does not start before the previously accepted job ends.*

Problem 2.8 (Interval Scheduling: Minimizing Maximum Lateness)**.** *In this interval scheduling problem, we are given n jobs, each of which has a length l_i and a deadline d_i (and can be scheduled at any time $t \geq 0$). We are then asked to complete the jobs one by one so that the maximum lateness of any single job is minimized.*

Example 2.9. *If we have two jobs with lengths and deadlines $l_1 = 4, d_1 = 3, l_2 = 6, d_2 = 2$ then we have the following options:*

1. *If we schedule job 1 before job 2 then job 1 will finish at time 4 and will be late by 1 and job 2 will finish at time 10 and will by late by 8.*

2. *If we schedule job 2 before job 1 then job 2 will finish at time 6 and will be late by 4 and job 1 will finish at time 10 and will be late by 7.*

To minimize the maximum lateness, it is better to schedule job 2 first as $max\{4, 7\} = 7 < \max\{1, 8\} = 8$. Note that this does not minimize the total lateness.

This problem can be stated more precisely using the following definition:

Definition 2.10. *Given a schedule $S = (i_1, \ldots, i_n)$ where $(i_1, \ldots, i_n)$ is a permutation of $[n]$, define $t(i_k) = \sum_{j=1}^{k} l_{i_j}$ to be the time at which job i_k is finished*

With this definition, our problem can be restated as follows. Find a schedule $S = (i_1, \ldots, i_n)$ where $(i_1, \ldots, i_n)$ is a permutation of $[n]$ such that $max_{j \in [n]}\{t(i_j) - d_{i_j}\}$ is minimized.

Algorithm 2.11 (Earliest deadline first)**.** *Take the schedule $S = (i_1, \ldots, i_n)$ such that $d_{i_1}, \ldots, d_{i_n}$ are in increasing order.*

Theorem 2.12. *The schedule $S = (i_1, \ldots, i_n)$ such that $d_{i_1}, \ldots, d_{i_n}$ are in increasing order minimizes $max_{j \in [n]}\{t(i_j) - d_{i_j}\}$.*

Proof. The idea behind the proof is as follows. Given an optimal schedule S', we can transform S' into S by exchanging pairs of jobs without increasing $max_{j \in [n]}\{t(i_j) - d_{i_j}\}$. This shows that S is optimal.

Lemma 2.13. *If $S' = (i'_1, \ldots, i'_n)$ is an optimal schedule, $j \in [n-1]$, and $d_{i'_j} > d_{i'_{j+1}}$ then the schedule S'' obtained by swapping i'_j and i'_{j+1} is also optimal.*

Proof. We make the following observations:

1. For all $j' \notin \{j, j+1\}$, $t(i'_{j'})$ is the same for both S' and S'' and thus $t(i'_{j'}) - d_{i'_{j'}}$ is the same for both S' and S''.
2. For S', $t(i'_j) = t(i'_{j-1}) + l_{i'_j}$ and $t(i'_{j+1}) = t(i'_{j-1}) + l_{i'_j} + l_{i'_{j+1}}$ (where we set $t(i'_0) = 0$)
3. For S'', $t(i'_j) = t(i'_{j-1}) + l_{i'_j} + l_{i'_{j+1}}$ and $t(i'_{j+1}) = t(i'_{j-1}) + l_{i'_{j+1}}$ (where we set $t(i'_0) = 0$)
4. For S', $\max\{t(i'_j) - d_{i'_j}, t(i'_{j+1}) - d_{i'_{j+1}}\} = t(i'_{j+1}) - d_{i'_{j+1}} = t(i'_{j-1}) + l_{i'_j} + l_{i'_{j+1}} - d_{i'_{j+1}}$ as $t(i'_{j+1}) > t(i'_j)$ while $d_{i'_{j+1}} < d_{i'_j}$.
5. For S'', $t(i'_j) - d_{i'_j} = t(i'_{j-1}) + l_{i'_j} + l_{i'_{j+1}} - d_{i'_j} < t(i'_{j-1}) + l_{i'_j} + l_{i'_{j+1}} - d_{i'_{j+1}}$ and $t(i'_{j+1}) - d_{i'_{j+1}} = t(i'_{j-1}) + l_{i'_{j+1}} - d_{i'_{j+1}} < t(i'_{j-1}) + l_{i'_j} + l_{i'_{j+1}} - d_{i'_{j+1}}$

Putting these observations together, the maximum lateness for S'' is less than or equal to the maximum lateness for S'. □

Using Lemma 2.13, whenever two adjacent jobs are out of order, we can swap them and put them in order. If we keep on doing this, the schedule will get closer to being in order with each step and this process can only terminate when everything is in order at which point we have reached the schedule S given by the greedy algorithm.

To make this argument rigorous, we use the lexicographical ordering on the deadlines.

Definition 2.14. *Given two schedules $S = (i_1, \ldots, i_n)$ and $S' = (i'_1, \ldots, i'_n)$, we say that $S < S'$ if there is a $k \in [n]$ such that $d_{i_k} < d_{i'_k}$ and for all $j \in [k-1]$, $i_j = i'_j$.*

Let $S' = (i'_1, \ldots, i'_n)$ be the optimal schedule which comes first in this lexicographical ordering, i.e. for any other optimal schedule S'', $S' < S''$. We claim that $S' = S$. To see this, assume $S' \neq S$. If so, there exists a $j \in [n-1]$ such that $d_{i'_j} > d_{i'_{j+1}}$. But then by Lemma 2.13, the schedule S'' obtained by swapping i'_j and i'_{j+1} in S' is also optimal and we have that $S'' < S'$, which contradicts how we chose S'. □

3 Review: Depth First Search and Breadth First Search

Problem 3.1 (Directed Connectivity)**.** *In the directed connectivity problem, we are given a directed graph $G = (V, E)$ and we are asked whether there is a path from some vertex s to another vertex t.*

Directed connectivity can be solved with the following kind of algorithm.

Algorithm 3.2 (Algorithms for directed connectivity)**.**

Stored data: We keep track of a set $S \subseteq V$ of vertices which we know are reachable from s. We also keep track of a set $E_{unexplored}$ of edges where for each $(u, v) \in E_{unexplored}$ we have reached u but may or may not have reached v.

Initialization: We start with $S = \{s\}$ and $E_{unexplored} = \{(s, v) : (s, v) \in E\}$.

Iterative step: At each step we remove an edge $(u, v) \in E_{unexplored}$ if possible and do the following:

1. *If this is not possible because $E_{unexplored}$ is empty then we output NO because we have already reached everything which is reachable from s and we have not reached t.*
2. *If $v \in S$ then we do nothing as v has already been reached before.*
3. *If $v \neq t$ and $v \notin S$ then we add v to S and add all of the edges $\{(v, w) : (v, w) \in E\}$ to $E_{unexplored}$.*
4. *If $v = t$ then we stop and output YES because we have reached our destination.*

This kind of algorithm solves directed connectivity and takes $O(|E|)$ time because each edge of G is added and removed from $E_{unexplored}$ at most once. However, to fully describe such an algorithm we need to describe how we add and remove edges from $E_{unexplored}$. The two standard choices for this are as follows:

1. If $E_{unexplored}$ is stored as a stack then this is the depth first search algorithm.
2. If $E_{unexplored}$ is stored as a queue then this is the breadth first search algorithm.

4 Dijkstra's Algorithm for Shortest Path

Problem 4.1 (Shortest Path Problem)**.** *In the shortest path problem, we are given the edge lengths of a directed graph G on n vertices (which must be non-negative). We are then asked to find a path of minimum length from some vertex s to another vertex t.*

More precisely, we are given the following data:

1. *A directed graph $G = (V, E)$ togther with a starting vertex $s \in V$ and a destination vertex $t \in V$.*
2. *For each edge $e = (u, v) \in E$, we are given its length $l_e = l_{uv}$ which must be non-negative.*

We are then asked to output a path P from s to t such that $\sum_{e \in E(P)} l_e$ is minimized.

Algorithm 4.2 (Dijkstra's Algorithm)**.** *Dijkstra's algorithm finds a path of minimum length from s to t by iteratively finding the paths of minimum length from s to every other vertex in G. To do this, Dijkstra's algorithm keeps track of candidate paths to new locations and then takes the shortest candidate path, which is guaranteed to be correct.*

More precisely, Dijkstra's algorithm can be described as follows.

Stored data: We keep track of the following:

1. *A set $S \subseteq V$ together with distances $\{d_v : v \in S\}$ where for all $v \in S$, d_v is the length of the shortest path P_v from s to v. For each $v \in S \setminus \{s\}$, we also keep track of the last edge (u, v) of P_v. Working backwards, this is sufficient to reconstruct P_v.*
2. *A set of edges $E_{candidate}$ together with distances $\{d_e : e \in E_{candidate}\}$ where for all $e = (v, w) \in E_{candidate}$, $d_e = d_v + l_e$ is the length of the path formed by taking the shortest path from s to v and adding the edge e.*

Initialization: We start with $S = \{s\}$, $d_s = 0$, $E_{candidate} = \{(s, v) : (s, v) \in E\}$, and $d_e = l_e$ for every edge $e = (s, v) \in E_{candidate}$

Iterative step: At each step we remove the edge $e = (v, w) \in E_{candidate}$ which has the smallest d_e and do the following:

1. *If this is not possible because $E_{candidate}$ is empty then we output that t is unreachable from s because we have already found the shortest paths to every vertex which is reachable from s and we have not reached t.*
2. *If $w \in S$ then we do nothing as we have already found a path of equal or smaller length to w.*
3. *If $w \neq t$ and $w \notin S$ then we add w to S, take $d_w = d_e = d_v + l_{vw}$, note that (v, w) was the last edge used to reach w, and add all of the edges $\{(w, x) : (w, x) \in E\}$ to $E_{candidate}$ with distances $d_{e'} = d_w + l_{e'}$.*
4. *If $w = t$ then we add t to S, take $d_t = d_e = d_v + l_{vt}$, and note that (v, t) was the last edge used to reach t. We then stop and output d_t as we have found the shortest path from s to t.*

Theorem 4.3. *Dijkstra's algorithm finds a path from s to t of minimum length and can be implemented in $O(|E|log(|E|))$ time.*

Proof. To prove that Dijkstra's algorithm finds the minimum length of a path from s to t, we prove the stronger statement that for all $v \in S$, d_v is the minimum length of a path from s to v.

Lemma 4.4. *We always have that for all $v \in S$, d_v is the minimum length of a path from s to v.*

Proof. We prove this by induction. The base case when $S = \{s\}$ is trivial as the shortest path from s to s has length 0. For the inductive step, assume that this statement holds when $|S| = k$ and consider what happens when we add the next vertex w to S via an edge (v, w) where $v \in S$. We claim that $d_w = d_v + l_{vw}$ is the minimum length of a path from s to w. To see this, assume that there is another path P' from s to w which has smaller length. Let w' be the first vertex on this path which is not in S and let (v', w') be the edge used to reach w'.

Observe that by the inductive hypothesis, the length of the part of P' between s and v' is at least $d_{v'}$. Moreover, we must have that $d_{v'} + l_{v'w'} \geq d_v + l_{vw} = d_w$ because otherwise Dijkstra's algorithm would have taken the edge (v', w') and the vertex w' rather than (v, w) and w. Thus, the length of the part of P' between s and w' is at least d_w, which contradicts the assumption that the length of P' is less than d_w.

Note that for this argument to work, it is important that the edge weights are non-negative as otherwise we could have that $d_{v'} + l_{v'w'} < d_v + l_{vw} = d_w$ even though $d_{v'} > d_v$. We will discuss how to handle negative weights later in the course. □

To implement Dijkstra's algorithm, we can use a priority queue (see below) to store the candidate edges (v, w) and distances d_e. Each edge e is added at most once and removed at most once, so we perform at most $2|E|$ operations on this priority queue. This priority queue never has more than $|E|$ elements, so each operation takes time $O(log(|E|))$. Thus, the total time required is $O(|E|log(|E|))$ □

Remark 4.5. *Instead of storing the edges of G in the priority queue, we can instead store the unreached vertices of G in the priority queue (together with the length of the shortest path to the vertex we've seen so far and the last edge of this path). Whenever we encounter a shorter path to an unreached vertex w, we decrease the weight of w.*

Using this approach, Dijkstra's algorithm can actually be implemented in time $O(|E|+|V|log(|V|))$ by using Fibonacci heaps to implement the priority queue. This is slightly faster in theory but is generally slower in practice because Fibonacci heaps are somewhat complicated.

4.1 Implementing Priority Queues

A priority queue is a data structure which stores elements together with their weights and supports the following operations:

1. Insert: Adds a new element to the priority queue.
2. Delete min: Removes the element with the smallest weight.

Some other common operations which may be implemented are as follows:

1. Peek: Return the element with the minimum weight without removing it.
2. Change weight: Given a pointer to an element, change its weight.
3. Merge: Merge two priority queues into one.

One way to implement a priority queue is with a balanced binary tree where each node has a smaller weight than its children. Using a balanced binary tree, the peek operation just needs to look at the root and the insert, delete min, and change weight operations can be implemented as follows.

1. Adding an element: To add an element, we place it in the next available spot. This may break the invariant that every parent is less than or equal to its children as this element may be smaller than its parent. To fix this, as long as this element is smaller than its parent, we swap the element and its parent.

2. Delete min: In order to remove the minimum element of the priority queue, we swap the element at the root of the tree with the element in the last occupied position of the tree and remove it. This may break the invariant that every parent is less than or equal to its children as the element which was moved to the root of the tree may be larger than one or both of its children. To fix this, as long as this element is larger than one or both of its children, we swap it with its smaller child.

3. Change weight: Given a pointer to an element, we can change its weight and then restore the invariant that each node has a smaller weight than its children in the same way as before. In particular, if the weight of the element is decreased then as long as this element is smaller than its parent, we swap the element and its parent. If the weight of the element is increased then as long as this element is larger than one or both of its children, we swap it with its smaller child.

A nice way to implement a balanced binary tree on N elements is as follows. We use an array with indices $1, \ldots, N$. We take index 1 to be the root. For each index i, we take the left child of i to be the index $2i$ and we take the right child of i to be the index $2i + 1$.

Remark 4.6. *The reason Fibonacci heaps are slightly faster (in theory) for Dijkstra's algorithm is because they can decrease the weight of an element in $O(1)$ time.*

5 Kruskal's Algorithm and Prim's Algorithm for finding Minimum Spanning Trees

Definition 5.1. *Given an undirected graph $G = (V, E)$, we say that $T \subseteq E$ is a spanning tree of G if (V, T) is a tree (i.e. (V, T) is connected and contains no cycles).*

Problem 5.2. *The minimum spanning tree problem is as follows:*

Input: *An undirected graph $G = (V, E)$ together with a length l_e for every edge $e \in E$.*

Output: *A spanning tree T of G such that $\sum_{e \in T} l_e$ is minimized.*

For simplicity, we assume that G is connected (as otherwise G does not have any spanning trees) and that all of the edges of G have distinct lengths. If G has multiple edges which have the same length, we can add a small perturbation to the edge lengths to make them all distinct and then return the resulting minimum spanning tree.

Algorithm 5.3 (Kruskal's Algorithm)**.** *Kruskal's algorithm first sorts the edges E in increasing order of their length. Kruskal's algorithm then considers the edges one by one and accepts each edge which does not form a cycle with the edges which were previously accepted.*

More precisely, Kruskal's algorithm can be described as follows:

Stored data: We keep track of a set of edges T and the set $E_{remaining}$ of the remaining edges of G which we have not yet considered. $E_{remaining}$ will always be sorted by length.

Initialization: We start with $T = \{\}$ and $E_{remaining} = E$ (sorted by length).

Iterative step: We remove the first edge e from $E_{remaining}$ (which will be the shortest edge we have not yet considered). If $T \cup \{e\}$ contains a cycle then we discard e and continue. Otherwise, we accept e by adding e to T. We stop when T is a spanning tree (in which case T will be the minimum spanning tree) or when $E_{remaining}$ is empty (in which case the original graph was not connected and there is no spanning tree).

Algorithm 5.4 (Prim's Algorithm)**.** *Prim's algorithm is similar to Dijkstra's algorithm. Prim's algorithm starts at a vertex s and then builds a tree out from s by iteratively accepting the shortest edge which leads to a new vertex.*

More precisely, Prim's algorithm can be described as follows:

Stored data: We keep track of the following:

1. *The set of edges T in our current tree and the set of vertices S in our current tree.*
2. *A set of edges $E_{candidate}$ together with their lengths l_e.*

Initialization: We start with $S = \{s\}$, $T = \{\}$, $E_{candidate} = \{\{s, v\} : \{s, v\} \in E\}$

Iterative step: At each step we remove the shortest edge $e = \{v, w\} \in E_{candidate}$ and do the following:

1. *If this is not possible because $E_{candidate}$ is empty then we output that there is no spanning tree because G is not connected.*
2. *If $w \in S$ then we do nothing as w is already part of the tree T.*
3. *If $w \notin S$ then we add w to S, add $e = \{v, w\}$ to T, and add all of the edges $\{\{w, x\} : \{w, x\} \in E\}$ to $E_{candidate}$ with their lengths l_e. If $S = V$ then we stop and output T.*

As we will see, both Kruskal's algorithm and Prim's algorithm give us the minimum spanning tree. The fundamental reason for this is that there is a nice characterization of the edges of the minimum spanning tree.

Definition 5.5. *Given a graph $G = (V, E)$ with edge lengths which are all distinct, define $E_{redundant}$ to be the set of edges e such that G contains a cycle C such that e is the longest edge in C, i.e.*

1. $e \in E(C)$
2. $\forall e' \in E(C) \setminus \{e\}(l(e') < l(e))$

Theorem 5.6. *If all of the lengths of the edges of G are distinct then there is a unique minimum spanning tree T of G with edges $E(T) = E(G) \setminus E_{redundant}$.*

To prove this theorem, we need the following lemma which says that it is sufficient to consider cycles where only one edge is in $E_{redundant}$.

Lemma 5.7. *Given a graph $G = (V, E)$ with edge lengths which are all distinct, if $e \in E_{redundant}$ then G contains a cycle C such that e is the longest edge in C and all other edges of C are not in $E_{redundant}$, i.e.*

1. $e \in E(C)$

2. $\forall e' \in E(C) \setminus \{e\}\,(l(e') < l(e)$ *and* $e' \notin E_{redundant})$

Proof. Assume this lemma is not true and let e be the shortest edge such that $e = \{u, v\} \in E_{redundant}$ and there is no cycle C in G such that

1. $e \in E(C)$
2. $\forall e' \in E(C) \setminus \{e\}\,(l(e') < l(e)$ and $e' \notin E_{redundant})$

If so, then take $E' = \{e' \in E : l(e') < l(e), e' \notin E_{redundant}\}$ and take $G' = (V, E')$. Observe that u and v are disconnected in G' as otherwise we would have such a cycle C. However, since $e \in E_{redundant}$, there is a cycle C' in G such that

1. $e \in E(C')$
2. $\forall e' \in E(C') \setminus \{e\}(l(e') < l(e))$

Let $w_1, \ldots, w_m$ be the path from $w_1 = u$ to $w_m = v$ in C' which does not use the edge $\{u, v\}$ and let j be the minimum index such that w_1 is not connected to w_j in G' (such a j must exist because w_1 and w_m are disconnected in G'). w_1 is connected to w_{j-1} in G' so w_{j-1} must be disconnected from w_j in G' and thus $(w_{j-1}, w_j) \notin E(G')$. Since $l_{(w_{j-1},w_j)} < l_e$, this implies that $e' = (w_{j-1}, w_j) \in E_{redundant}$.

Moreover, since w_{j-1} must be disconnected from w_j in G' there is no cycle C'' such that

1. $e' = \{w_{j-1}, w_j\} \in E(C'')$
2. $\forall e'' \in E(C'') \setminus \{e'\}, l(e'') < l(e')$ and $e'' \notin E_{redundant}$

However, this contradicts how we chose e. □

With this lemma in hand, we can now prove Theorem 5.6

Proof of Theorem 5.6. Let T be a minimum spanning tree of G and assume that there is an edge e such that $e \notin E_{redundant}$ and $e \notin E(T)$. Since T is a spanning tree of G, $E(T) \cup \{e\}$ contains a cycle C. Since $e \notin E_{redundant}$, this cycle C must contain an edge e' such that $l(e') > l(e)$. Now observe that $T' = (T \setminus \{e'\}) \cup \{e\}$ is also a spanning tree and has smaller total length than T, which contradicts the assumption that T is a minimum spanning tree.

Conversely, if $e \in E_{redundant}$ then by Lemma 5.7 there is a cycle C such that e is the longest edge in C and all other edges $e' \in E(C)$ are not in $E_{redundant}$. By the above argument, $E(C) \setminus \{e\} \subseteq E(T)$ which implies that $e' \notin E(T)$ as otherwise T would contain a cycle. □

Theorem 5.8. *Kruskal's algorithm and Prim's algorithm both give the minimum spanning tree.*

Proof. To show that Kruskal's algorithm gives the minimum spanning tree T, we need to show that it rejects every edge in $E_{redundant}$ and accepts every edge in $E(G) \setminus E_{redundant}$. We can prove this by induction.

The base case is trivial as Kruskal's algorithm will always accept the edge which has the smallest length. For the inductive step, assume that Kruskal's algorithm is correct on the edges so far and consider the next edge e. There are two cases to consider:

1. If $e \notin E_{redundant}$ then $T_{current} \cup \{e\}$ does not contain a cycle because $T_{current} \cup \{e\} \subseteq T$ and T contains no cycles. Thus, Kruskal's algorithm will correctly accept e.

2. If $e \in E_{redundant}$ then by Lemma 5.7, there exists a cycle C containing e such that every other edge of C is not in $E_{redundant}$ and has smaller length than e. By the inductive hypothesis, all of these edges will be in $T_{current}$ so $T_{current} \cup \{e\}$ contains C and thus Kruskal's algorithm will correctly reject e.

To show that Prim's algorithm gives the minimum spanning tree, observe that Prim's algorithm only accepts an edge e if e is the shortest edge between S and $V \setminus S$ where S is the set of vertices which have been reached so far. Such an edge e can never be in $E_{redundant}$ because any cycle containing e must contain another edge between S and $V \setminus S$ and any such edge is longer than e. Thus, Prim's algorithm never accepts an edge in $E_{redundant}$.

To see that Prim's algorithm accepts all edges which are not in $E_{redundant}$, just as with Kruskal's algorithm, observe that at the time an edge $e \notin E_{redundant}$ is considered, $T_{current} \subseteq T$ and thus $T_{current} \cup \{e\} \subseteq T$ so e does not form a cycle. Thus, e must go to a vertex $w \notin S$ so e is accepted. □

Remark 5.9. *Prim's algorithm can be implemented in the same way as Dijkstra's algorithm and thus takes $O(|E|log(|E|))$ time. To implement Kruskal's algorithm, we first need to sort the edges by length, which takes $O(|E|log(|E|))$. We also need a way to test whether $T \cup \{e\}$ contains a cycle. This can be done using the union find data structure in $O(\alpha(n))$ amortized time per query (i.e. some queries may take longer but the time required for n queries is $O(n\alpha(n))$). Here $\alpha(n)$ is the inverse Ackermann function which is not quite constant but grows incredibly slowly. For all practical purposes, $\alpha(n) \leq 4$.*