CS 577: Introduction to Algorithms Fall 2015

Greed

Instructor: Dieter van Melkebeek Scribe: Andrew Morgan

DRAFT

A huge number of problems in algorithms development can be described in the following generic way: The problem instance is a system with a bunch of components and each of these components can be in some set of states. There is also an objective function, which looks at the settings of each of the components, and tells us the quality of the overall configuration. The problem is to assign a setting to each of the components in order to optimize this objective function.

A simple solution to this general sort of problem is to enumerate (or 'brute-force') all of the possible configurations, and pick the best one. This can take a very, very long time.

On the other hand, sometimes a seemingly naïve approach can be made to work: we simply enumerate the components, one at a time, and put each one in a state that, ignoring the consequences for future components, maximizes the objective. This approach has two parameters: the *order* in which the components are processed, and the *strategy* we employ for each component. For some problems, the right choice of order and strategy can lead to an extremely simple and efficient algorithm. Algorithms fitting this paradigm are said to be 'greedy'.

As a simple illustrative example, consider the following variant of the Knapsack Problem: Given n items of weights $w_1, w_2, \ldots, w_n$, and a capacity W of the total weight we can carry in a knapsack, choose a subset of the items which fit into the knapsack which has the most items in it. In this case, the components correspond to the items, and each component can be in a state of "put in the knapsack" or "don't put in the knapsack". The quality of a configuration is either the number of items in the knapsack, if they fit, or else zero if the chosen subset does not fit into the knapsack.

If we try the brute-force approach, this requires considering all 2^n subsets of items. When there are even only 250 items to consider, this approach already has us considering a few subsets per atom in the observable universe.

But this problem has a simple greedy algorithm:

1. *Order* the items in order of weight, with the lightest item first.
2. Use the following *strategy* on each item in order: if the item fits into the rest of the knapsack, we will choose to put it into the knapsack; otherwise, we will leave it out.

This approach only takes time $\Theta(n \log(n))$ to sort, and then only $\Theta(n)$ time to decide which items to take, meaning an overall $\Theta(n \log(n))$ running time. This is a tremendous speedup over the brute-force algorithm.

This approach also turns out to correctly find a way of filling the knapsack with the maximum number of items. Of course, we would have to prove this, and we will in Section 2.1. Unfortunately, while greedy algorithms are often very simple to implement, they do tend to require the most creative thinking in order to prove correctness.[1] However, we will cover a couple general

[1] Of course, some much more sophisticated algorithms will require more creativity than we will see here, but this statement is generally true when comparing greedy approaches to the remaining algorithmic paradigms we will see in this algorithms class.

strategies which help prove correctness for most greedy algorithms. These strategies are usually called "greedy-stays-ahead arguments" and "exchanges arguments". Similar to how recursion trees provided a nice scaffold for understanding divide and conquer algorithms, but didn't completely solve any specific problem, the strategies will only provide scaffolding—it is up to the algorithm designer to fill in the problem-specific details. To help see how the scaffolding fits together, we will present the approaches by means of examples.

1 Greedy Stays Ahead

1.1 Interval Scheduling

Our first example of a problem with a greedy solution will be that of interval scheduling. Imagine a receptionist has been taking down reservations for a conference room from various groups of people. However, she has made a grave mistake: she forgot to check for conflicts. Having just realized her mistake, she now wants to notify as few groups as possible that their reservations have been canceled or will need to be changed. Fortunately for this receptionist, there is a simple greedy strategy that will find the best solution. Before we go tell her, we should first describe it to ourselves and then formally prove its correctness.

Here is the interval scheduling problem, formally specified:

Input: An array of intervals $I_i = [s_i, e_i)$ for $i = 1 \ldots n$

Output: A maximum-sized subset of intervals for which every pair of distinct intervals is disjoint.

The output in this case corresponds to the reservations that the receptionist does not cancel; since every reservation is either canceled or not, maximizing this is the same as minimizing the number of cancellations.

One aspect of the greedy strategy is straightforward, namely the strategy that our algorithm will use once we have decided on a good order on the intervals. A good general rule for constructing these strategies is to pretend that the interval being considered is the *last* one in the order. For us, the strategy we will use is simply to take every interval we see, except when doing so conflicts with intervals we have already taken. This turns out to work for an appropriate order.

Unfortunately for the receptionist, a good choice of order is not entirely obvious. The following ideas were proposed in class:

1. Size of the interval (smallest first)
2. Start time (earliest first)
3. Start time (latest first)
4. Finish time (earliest first)
5. Finish time (latest first)
6. Number of conflicts (least first)

If we imagine 'flipping' all the intervals around *i.e.*, considering the interval $[s_i, e_i)$ instead as $[-e_i, -s_i)$, then the answer will be unaffected. However, sorting using rule (2) results in the same

Figure 1: Interval Scheduling Sorting Policy Counterexamples

(Forthcoming)
(a) Smallest interval size first fails

(Forthcoming)
(b) Earliest start time first fails

(Forthcoming)
(c) Least conflicts first fails

order as not flipping and using rule (5). Similarly, rules (3) and (4) are the same up to flipping. So let's ignore rules (3) and (5) for now.

Of the four ideas remaining, only one works. Using (1) to sort, using our strategy on the instance depicted in Figure 1a will clearly result in the wrong answer. Using (2) to sort, Figure 1b will also yield a wrong answer. Using (6) to sort, things seem better, but Figure 1c will again result in the wrong answer.

Using (4) to sort, however, will actually work. A pseudocode implementation (which shows how to check for conflicts efficiently) is given in Algorithm 1.

Algorithm 1

Input: $s[0, \ldots, n-1], e[0, \ldots, n-1]$, where the interval $I_i = [s[i], e[i])$
Output: A maximum-size set $S \subseteq \{0, 1, \ldots n-1\}$ so that $I_i \cap I_j = \varnothing$ for distinct $i, j \in S$

```
1:  procedure INTERVAL-SCHEDULING(s, e)
2:      Intervals[0, ..., n − 1] ← array with Intervals[i] = i
3:      Sort Intervals so that e[Intervals[i]] ≤ e[Intervals[i + 1]] for each i
4:      LastConflict ← −∞
5:      S ← ∅
6:      for i = 0 ... n − 1 do
7:          if s[i] ≥ LastConflict then
8:              LastConflict ← e[i]
9:              S ← S ∪ {i}
10:     return S
```

We can prove the correctness of Algorithm 1 using the strategy of *greedy stays ahead*. The general strategy with greedy-stays-ahead is to figure out two quantities: a notion of time and a notion of 'ahead'. If these two things are chosen appropriately, then a greedy-stays-ahead proof starts by showing (usually by induction) that at every time step, greedy is 'ahead' of any other solution. Then an argument is made to say that, if greedy is 'ahead' of every other solution at the last time step, then greedy has found an optimal solution.

For the problem of interval scheduling, there are actually (at least) two ways to approach this, and we will present both of them.

Greedy Stays Ahead, Proof 1 Let's first get an intuitive idea for why greedy is correct. We would like to show that greedy finds a set of conflict-free intervals of maximum size. Let's suppose that $I_1, I_2, \ldots, I_n$ are already sorted in order of increasing end time.

We'll start simple, and consider the case where there exists a conflict-free (but not necessarily optimal) solution that contains only one interval. In this case, it's clear that greedy will find at least as good of a solution: it always includes I_1.

Moving on, let's now suppose that there exists a conflict-free solution with two intervals; say I_i and I_j with $i < j$. These intervals don't conflict, so I_i has to be entirely before I_j, *i.e.*, $e_i \leq s_j$. In

this case, greedy will again find at least as good of a solution. Greedy chooses I_1, and $e_1 \leq e_i \leq s_j$, so when greedy considers I_j, it will either select it (because it doesn't conflict with I_1) or it must have already selected some second interval.

Finally, let's consider the case that some solution has three intervals, say I_i, I_j, I_k with $i < j < k$. Again, none of these intervals conflict, so we know that $e_i \leq s_j$ and $e_j \leq s_k$. We know that greedy will choose I_1, and some interval I_ℓ to be its second interval. This is the same argument as before applied to the pair I_i and I_j. But we also know by that argument that $\ell \leq j$. So in particular we know that $e_\ell \leq e_j$, thus $e_\ell \leq s_k$, and so I_ℓ and I_k don't conflict. This means we can again conclude that greedy has to choose some third interval: either it will choose its third interval before I_k, or else it will choose I_k.

This process continues for optimal solutions of size four, five, and so on. The argument is essentially the same each time, and in particular, we have an inductive argument! It's not as clear as it should be though, so let's formalize it some more.

Let $S \subseteq \{1, 2, \ldots, n\}$ index any conflict-free set of intervals, and let $G \subseteq \{1, 2, \ldots, n\}$ index the set of intervals that greedy chooses. Our induction is on the size of S, $|S|$, and we want to show that greedy always chooses at least as many elements as are in S, *i.e.*, $|G| \geq |S|$. But we also needed this idea that the last element of S comes after the $(|S|-1)$-th element of G in order to argue that G picked an $|S|$-th interval.

To formalize that, let's define the function $f_S(i)$ for $i = 1, 2, \ldots, |S|$ to be the ending time of the i-th interval of S. For example, if $S = \{1, 2, 5, 6, 7\}$, then $f_S(4) = e_6$, the ending time of I_6, and $f_S(6)$ is undefined since S has only five elements. Let $f_G(i)$ be similarly defined for the greedy schedule, for $i = 1, 2, \ldots, |G|$.

Then we can say that greedy 'stays ahead' in the following sense:

Claim 1. *For every subset $S \subseteq \{1, 2, \ldots, n\}$ that indexes a conflict-free set of intervals, $|G| \geq |S|$, and $f_G(|S|) \leq f_S(|S|)$.*

It's clear that Claim 1 implies the optimality of the greedy strategy: it applies to every choice S of conflict-free intervals, and in particular to any optimal choice. Since $|G| \geq |S|$ even in these cases, it follows that the greedy solution is at least as good as any optimal solution.

Let's now prove our claim by translating our previous argument into a formal proof by induction.

Proof. As stated, the proof is by induction on $|S|$.

$|S| = 1$: Greedy has not added any intervals before I_1, so adding I_1 cannot cause a conflict. Thus it must be the case that 1 is in G, and so $f_G(1) = e_1$.

We know that $e_1 \leq e_j$ for every $j = 1, \ldots, n$, and we know that these are all the values $f_S(1)$ can be. Thus we can conclude that $f_G(1) = e_1 \leq f_S(1)$.

$|S| > 1$: Let I_k be the interval of S with latest end time, so that $f_S(|S|) = e_k$. The assumption that S has no conflicts translates to the fact that $f_S(|S|-1) \leq s_k$.

Let S' be S without k, *i.e.*, $S - \{k\}$. Since $S' < |S|$ and S' has no conflicts, we can apply our inductive hypothesis to S'. This tells us $|G| \geq |S'|$ and $f_G(|S'|) \leq f_{S'}(|S'|)$. Since I_k is the last interval in S, we know that $f_{S'}(|S'|) = f_S(|S|-1)$. Thus our inductive hypothesis tells us that $|G| \geq |S| - 1$, and furthermore $f_G(|S|-1) \leq f_S(|S|-1)$.

Combining the conclusions of the last two paragraphs, we can conclude that $f_G(|S|-1) \leq s_k$. Thus, we can conclude that greedy must have chosen an $|S|$-th interval: either greedy added

an interval $I_{k'}$ with $k' < k$ as its $|S|$-th interval, or else greedy added I_k. This is because the inequality $f_G(|S|-1) \leq s_k$ shows that I_k does not conflict with any of the first $|S|-1$ intervals in G. Thus $|G| \geq |S|$. Moreover, this argument shows that greedy had chosen its $|S|$-th interval *no later* than when considering I_k. It follows that $f_G(|S|) \leq e_k$. □

The way this proof fits into the greedy-stays-ahead strategy is as follows: The notion of 'time' was measured by 'number of intervals in the solution'. The notion of 'ahead' was the inequality $|G| \geq |S|$, but also the bound $f_G(|S|) \geq f_S(|S|)$. It was quite easy to establish that greedy did well in the first time step—when only one interval was in the solution. We also showed that, with each successive increase in an alternative solution S, greedy 'stayed ahead' of S by having a solution of at least the same size *and* whose $|S|$-th interval ended no later than the last interval of S. From this we concluded that, even at the last time step—when the alternative solutions S were optimal—greedy still performed at least as well as every solution. This implied that greedy was optimal overall.

Before moving on, note that our notion of 'ahead' was a bit stronger than was needed for that last step. It sufficed to simply prove that $|G| \geq |S|$ for every other conflict-free S, but we proved this in addition to the fact that $f_G(|S|) \leq f_S(|S|)$. The reason for this stronger notion of ahead was that it aided in our inductive step.

Greedy Stays Ahead, Proof 2 Let's now move on to the second greedy-stays-ahead proof of correctness for our greedy algorithm. Let S again denote some non-conflicting subset of intervals.

We viewed our greedy approach as first fixing some order on the intervals, enumerating the intervals in this order, and then making decisions about the intervals as we process them. We can interpret S as having fixed the same order on the intervals, and then employing a *different* decision-making strategy to select intervals. In particular, S's strategy is simply to select exactly the intervals in S. Thus the main difference between the greedy strategy and S will just be the decisions they make while processing the intervals in greedy's order.

This almost immediately gives us a notion of 'time' for our greedy-stays-ahead approach. The i-th time step will simply consider the partial solutions that the greedy algorithm and S have after considering the first i intervals. A natural choice for 'ahead' then is simply to say that, at each time step, greedy's partial solution is at least as large as the partial solution from S.

This combination turns out to work for our greedy algorithm for interval scheduling. Since we already know that the algorithm is correct, we won't belabor the intuition behind the proof; instead we'll jump straight to the formal claim and proof.

First, we need some notation. Let's again assume for simplicity that the input intervals $I_1, I_2, \ldots, I_n$ are already sorted in order of increasing end time. Again, let $G \subseteq \{1, 2, \ldots, n\}$ denote the subset selected by the greedy algorithm. For $t = 1, 2, \ldots, n$, let $c_S(t) = |S \cap \{1, 2, \ldots, t\}|$ be the number (or 'count') of indices in S which are at most t, and let $c_G(t) = |G \cap \{1, 2, \ldots, t\}|$ be the same with respect to the greedy strategy instead of S. These represent the number of intervals that S and G (respectively) have in their partial solutions after considering the t-th interval. We can then formulate our claim that 'greedy stays ahead' as follows:

Claim 2. *For every* $t = 1, \ldots, n$, $c_G(t) \geq c_S(t)$.

With this claim in hand, it is straightforward to prove the correctness of the greedy algorithm. Since $c_S(n) = |S|$ and $c_G(n) = |G|$, Claim 2 specialized to $t = n$ shows that $|G| \geq |S|$. It then follows (as in the first proof) that greedy correctly finds an optimal solution.

Let's now prove Claim 2.

Proof. As promised, we will use (strong) induction.

$t = 1$: When $t = 1$, $c_G(t) = 1$ and $c_S(t) \leq 1$, so $c_G(t) \geq c_S(t)$.

$t > 1$: There are three cases to consider based on whether $t \in S$ and whether $t \in G$.

$t \notin S$ In this case, $c_G(t) \geq c_G(t-1)$ and $c_S(t) = c_S(t-1)$. Since we know $c_G(t-1) \geq c_S(t-1)$ by induction, the claim follows since

$$c_G(t) \geq c_G(t-1) \geq c_S(t-1) = c_S(t)$$

$t \in G$ **and** $t \in S$ In this case, $c_G(t) = c_G(t-1) + 1$ and $c_S(t) = c_S(t-1) + 1$. Since we know $c_G(t-1) \geq c_S(t-1)$ by induction, the claim follows since

$$c_G(t) = c_G(t-1) + 1 \geq c_S(t-1) + 1 = c_S(t)$$

$t \notin G$ **and** $t \in S$ In this case, $c_G(t) = c_G(t-1)$ and $c_S(t) = c_S(t-1)+1$, so we can't apply the same general reasoning as in the last two cases. However, we can still say something: Let i_G and i_S be the largest elements of G and S, respectively, which are strictly less than t. These correspond to the most recently-added intervals before the t-th. Since $t \notin G$, it must be the case that the i_G-th interval conflicts with the t-th interval; otherwise the greedy algorithm would have included I_t in its solution. Since $t \in S$, it must be the case that the i_S-th interval does not conflict with the t-th interval; otherwise S would have a conflict. In particular, we can deduce that $i_G > i_S$.

So now let's back up to when we were consider the i_S-th interval. By the definition of i_S, we know that $c_S(i_S) = c_S(t) - 1$. Furthermore, since $i_G > i_S$, we know that $c_G(i_S) \leq c_G(t) - 1$. By induction we know that $c_G(i_S) \geq c_S(i_S)$, so with some algebra we have

$$c_G(t) \geq c_G(i_S) + 1 \geq c_S(i_S) + 1 = c_S(t)$$

which concludes the proof.

□

1.2 Dijkstra's Algorithm for Shortest Paths

One of the most common problems over graphs is that of finding the shortest path between a given pair of vertices. Formally, the (Single-Source, Single-Sink) Shortest Paths problem has the following specification:

Input: A directed graph $G = (V, E)$ on n vertices and m edges, a pair of vertices $s, t \in V$, and a non-negative edge-length function, $\ell : E \to [0, +\infty)$.

Output: The distance and a shortest path from s to t if there is a path from s to t, or else an indication that t is not reachable from s.

Note that we require the edge-length function ℓ to be non-negative. This is important for this problem to have a meaningful answer. In particular, if we allow negative edge lengths, then it's quite possible that a negative cycle will exist in the graph G. If we ask for a shortest path with s and t both on this cycle (among other situations), then this shortest path isn't well-defined. This

is because we can take any path from s to t, and add to the end of it a path around the cycle. Since the cycle has negative total length, the new path has smaller total length. This means that there is no single path from s to t with minimum total length. This in turn means our specification wouldn't be well-defined in these cases, which is an issue.

But even if we required the input graph to contain no negative cycles—in which case the shortest paths between connected vertices will always be defined—the greedy approach we will develop here still turns out to be flawed in the face of negative edge lengths. Later, we will see (slower) other algorithms for handling this situations; for now we will just cover the setting of non-negative edge lengths.

Shortest paths have an interesting property, namely they satisfy the following proposition:

Proposition 1. *Let $w_0, w_1, \ldots, w_k$ be a shortest path, and i and j be indices of this path with $i \leq j$. Then $w_i, w_{i+1}, \ldots, w_j$ is also a shortest path.*

Proof. We prove the contrapositive: suppose that $w_i, w_{i+1}, \ldots, w_j$ is not a shortest path. Let $w_i, v_1, v_2, \ldots, v_s, w_j$ be a shorter path from w_i to w_j. Then the path

$$w_0, w_1, \ldots, w_i, v_1, v_2, \ldots, v_s, w_j, w_{j+1}, \ldots, w_k$$

is shorter than the path $w_0, w_1, \ldots, w_k$. □

In other words, sub-paths of shortest paths are themselves shortest paths. This is important for shortest paths problems because it lets us represent shortest paths from a fixed vertex in a compact form. If we fix the source s, then we can represent a set of shortest paths to *every* vertex in $O(n)$ space in the following way: Define the array Parent to be indexed by the vertices of the graph. For each vertex v, Parent satisfies the property that Parent[v] is a vertex immediately before v in some shortest path from s to v. We can use Parent to recover a shortest path from s to v by simply backtracking: the last vertex is v, the second-to-last vertex is Parent[v], the third-to-last is Parent[Parent[v]], and so on, until Parent[$\ldots$Parent[v] $\ldots$] is s. We can therefore require the output of the shortest path problem to simply be the Parent array.

If we let $d(v)$ denote the length of any shortest path from s to v, then we can more precisely state the specification of the shortest path problem as follows:

Input: A directed graph $G = (V, E)$ on n vertices and m edges, a pair of vertices $s, t \in V$, and a non-negative edge-length function, $\ell : E \to [0, +\infty)$.

Output: If there is no path from s to t, then an indication of this fact. Otherwise, the value $d(t)$ and the Parent[$\cdot$] array such that Parent[t], Parent[Parent[t]], and so on are all defined, and represent a shortest path from s to t.

Greedy Approach As suggested by the topic of these notes, the algorithm we give will be greedy. We will want the 'components' to be the vertices v, and their 'states' will be the choices for Parent[v] and $d(v)$. The quality of a configuration is then the length of the shortest path from s to t. Thus a greedy algorithm would enumerate the vertices in some order, and assign a parent to each vertex in turn. Let's now try to develop some of the intuition behind our greedy algorithm.

From Proposition 1, we know that, if we have found a shortest path from s to v, then we have found a shortest path from s to every vertex on this shortest path to v. This suggests that finding

Figure 2: How to grow cuts: an example cut

(Forthcoming)

shortest paths with many edges in them will be harder than finding shortest paths with fewer edges. As an extreme example, we already know a shortest path from s to s: the path with no edges based at s must be a shortest path. This empty path has total length zero, while any path must have total length at least zero, because the length function, ℓ, has nonnegative values. For this reason, it makes sense to try to 'grow' our shortest paths out from s in some way.

This motivates our use of the notion of a cut: A *cut* is a subset $C \subseteq V$ of vertices. It is equivalently a pair of subsets (L, R), where $R = V - L$: the equivalence is simply by choosing $L = C, R = V - C$. Yet another way to formalize a cut is by the edges that cross it, *i.e.*, a cut (L, R) can also be regarded as the set of edges $E(L, R) \doteq \{(u, v) : u \in L, v \in R\}$. These are all equivalent notions, so we will use whichever one is most appropriate to the relevant context.

Cuts allow us to keep track of our 'growth'. In particular, the set of vertices to which we know shortest paths from s can be regarded as a cut. Once t is in this cut, we can stop. Similarly if the cut has no edges crossing it and t is not in the cut, then we know that there is no path from s to t, and again we can stop. Otherwise, we can use the cut to help us reason out a new vertex to add to the cut.

Let's suppose for now that we have a cut (L, R), such that we know shortest paths from s to every vertex in L. We would like to find a vertex in R to which we can infer a shortest path from s. A natural choice (following Proposition 1) is to consider vertices v in R which are on the *border* of the cut. More precisely, we want to consider vertices in R which are neighbors of vertices in L. If we can find a shortest path to any vertex in R, then we can find a shortest path to such a border vertex because of Proposition 1.

Another way to phrase this is that we want to grow our cut (regarded as a set of vertices) *along the edges in the cut* (now regarded as the edges crossing the cut). Figure 2 contains an example of this scenario, which is useful for understanding the following more abstract discussion.

Note that, for each choice of edge (u, v) in the (L, R) cut, we can get a short path from s to v by simply taking the shortest path from s to u, and then crossing the edge to v. For ease of notation, for edges (u, v) in the cut, let $P(u, v)$ denote a path formed this way. The length of $P(u, v)$ is simply $d(u) + \ell(u, v)$. It's not necessarily true that all of these $P(u, v)$'s will be shortest paths, but we do know that at least one of them must be a shortest path.

It is natural to suspect that, of these $P(u, v)$'s, a shortest one must be a shortest path. In fact, we can argue that this is indeed the case. Formally we have the following claim.

Claim 3. *Let (L, R) denote a cut with s in L. Let $u \in L, v \in R$ be so that (u, v) is an edge in G, and so that the length of $P(u, v)$, $d(u) + \ell(u, v)$, is minimized. Then*

Proof. Let P' be any path from s to v.

Since v is on the opposite side of the cut from s, P' must cross the (L, R) cut. In particular, P' has a sub-path which starts at s, travels along some path contained entirely on s's side of the cut (L), visits a vertex u', and then takes it final step to a vertex v' on the side of the cut opposite s (R). Let P'' represent this subpath. Clearly the length of P' is at least the length of P''—this follows from the fact that the edge lengths are nonnegative.

We claim that P'' has length at least the length of $P(u, v)$ and that this suffices to prove our claim.

For one, P'' can be no longer than $P(u', v')$. This is because $P(u', v')$ and P'' both end in the edge (u', v'), and because the rest of $P(u', v')$ is a shortest path. Thus the length of P'' is at least $d(u') + \ell(u', v')$.

Second, $P(u', v')$ is no longer than $P(u, v)$; this is because we chose $P(u, v)$ so that $d(u)+\ell(u, v)$ was minimal, and thus no larger than $d(u') + \ell(u', v')$.

Putting these two together, we have that $P(u, v)$ has length at most the length of P''. Since P' is at least the length of P'', we conclude that $P(u, v)$ has length at most the length of P'.

Since P' is any path from s to v, it follows that $P(u, v)$ must be a shortest path from s to v. □

This means our greedy strategy will be the following:

1. Start out with $L = \{s\}$, $R = V - L$, and $d(s) = 0$.
2. Repeat the following until there are no more edges in the (L, R) cut or t is in L:
   (a) Find the edge $(u, v) \in E(L, R)$ in the cut which minimizes the quantity $d(u) + \ell(u, v)$.
   (b) Set $d(v) \leftarrow d(u) + \ell(u, v)$. Set Parent$[v] = u$.
   (c) Remove v from R and add v to L.
3. Return $d(t)$ and Parent.

This is essentially the complete algorithm. We've already argued its correctness in the above discussion. Let's quickly see how efficient it is. Steps 1 and 3 are easy to implement. Step 2 is repeated at most once per vertex in the original graph, since L grows with each iteration. Step 2(b) will only take constant time per iteration. Step 2(c) also involves little work—if we assume that adding to and removing from sets can be done in constant time[2], then these operations can be done in constant time. Step 2(a) on the other hand is less trivial. If we implement it naïvely, by enumerating each edge of the graph, then one iteration of step 2(a) takes time linear in the number of edges. This would overall make the running time $O(n \cdot m)$. However, we can do a lot better with good data structures.

Speeding up the implementation The exact data structure that we will need will be a *priority queue*. We will cover priority queues in more depth later in these notes, but for now let's state their relevant properties and finish our algorithm for shortest paths.

A priority queue is a container (like a list or array, it stores multiple kinds of similar-type objects) which stores objects according to a value called their *key*. Keys need to be comparable in some way. For example, one could store some integers keyed by their value, or a set of clients keyed by their soonest project deadline. A priority queue implements the following operations:

Create: Given an array A of n elements and their keys, a priority queue can be created containing exactly the elements of A.

Insert: Given a new element x and key k, a priority queue inserts the element x with key k.

[2] If we used a 'set' represented in the usual way by a balanced binary tree, then these operations would take time logarithmic in the size of the set. However, we don't need the full power of a generic 'set' container. We can represent whether a vertex is in L or R by keeping an auxilliary array telling us to which set a vertex belongs. Moving a vertex from R to L and checking whether the edge (u, v) is in $E(L, R)$ can both be done with a constant number of array reads and writes.

Decrease-Key: Given an element x of the priority queue, and a new key value k' which is at most x's present key value, change x's key to be k'.[3]

Extract-Min: If the priority queue is empty, report an error. Otherwise, remove an element x whose key is minimal among all elements of the priority queue.

Later, we will give an implementation of a priority queue in which the Create operation can be done in time $\Theta(n)$, and the Insert and Extract-Min operations can be done in time $\Theta(\log(n))$, where n is the number of elements in the priority queue. For now, let's finish our presentation of our shortest paths algorithm.

The basic idea is to keep only border vertices in the priority queue, and to key the vertex v by the smallest value $d(u)+\ell(u,v)$ for which (u,v) is a cut edge. We'll also keep track, for each border vertex v, of the vertex u that gives the smallest value. In this way, step 2(a) above can essentially be replaced with a simple Extract-Min call to the priority queue. Step 2(c) gets more complicated since we have to update the priority queue. But this again turns out to be pretty straightforward; the details are given in the pseudocode in Algorithm 2.

Algorithm 2

Input: $G=(V,E), \ell : E \to [0,+\infty), s$, the input graph, edge-lengths, and source
Output: Parent, d, the information about shortest paths from s

```
1:  procedure DIJKSTRA(G, ℓ, s)
2:      d(v) ← ∞ for v ∈ V
3:      Parent ← array indexed by V, initialized to none
4:      BeenQueued ← array indexed by V, initialized to false
5:      Q ← new, empty priority queue
6:      d(s) ← 0
7:      Q.INSERT(s,d(s))
8:      BeenQueued[s] ← true
9:      while Q is not empty do
10:         v ← Q.EXTRACT-MIN
11:         for u, a neighbor of v do
12:             if d(v) + ℓ(v,u) ≥ d(u) then
13:                 continue
14:             Parent[u] ← v
15:             d(u) ← d(v) + ℓ(v,u)
16:             if ¬BeenQueued[u] then
17:                 Q.INSERT(u, d(u))
18:                 BeenQueued[u] ← true
19:             else
20:                 Q.DECREASE-KEY(u, d(u))
```

Let's end with a new running time analysis. The setup (Lines (2)–(8)) can be done in time $O(n)$. The insert priority queue operation is used at most once per vertex of G. Accordingly,

[3] Not all priority queue implementations have a Decrease-Key operation. Nevertheless, there's a slight variation on the pseudocode in Algorithm 2 which yields the same asymptotic performance, and the constant factor involved is only at most twice as big when a priority queue is implemented as a heap. Can you find it?

extract-min is used at once once per vertex of G. Finally, the reduce-key operation is used at most once per edge in G. The remaining operations take total time $O(n+m)$. Thus the overall running time is $O(n+m+(\alpha+\beta)n+\gamma m)$, where α, β, and γ respectively denote the cost of the insert, extract-min, and reduce-key operations on a priority queue. When we give an implementation of a priority queue in the next section, we will achieve the parameters $\alpha, \beta, \gamma = O(\log(m))$. This leads to a final running time of $O((n+m)\log(m))$.

1.3 Priority Queues

We now turn our attention to the priority queue data structure. In Dijkstra's algorithm, priority queues allow us to keep track of all the "next" vertices to consider for growing the cut. The essential functionality we needed was to be able to insert vertices with their distances across the cut into the queue, and repeatedly extract the vertex with minimal distance-across-the-cut. In general, priority queues provide a way of maintaining a dynamic set of objects in such a way that arbitrary objects can be added quickly, and the "best" object can be popped off efficiently.

The specific interface of a priority queue was stated above, but we repeat it here for completeness. A prriority queue is a data structure that stores objects, and has associated to each object a key. Priority queues provide the following operations:

Create: Given an array A of n elements and their keys, a priority queue is created containing exactly the elements of A.

Insert: Given a new element x and key k, the priority queue inserts the element x with key k.

Decrease-Key: Given an element x of the priority queue, and a new key value k' which is at most x's present key value, change x's key to be k'.

Extract-Min: If the priority queue is empty, report an error. Otherwise, remove an element x whose key is minimal among all elements of the priority queue.

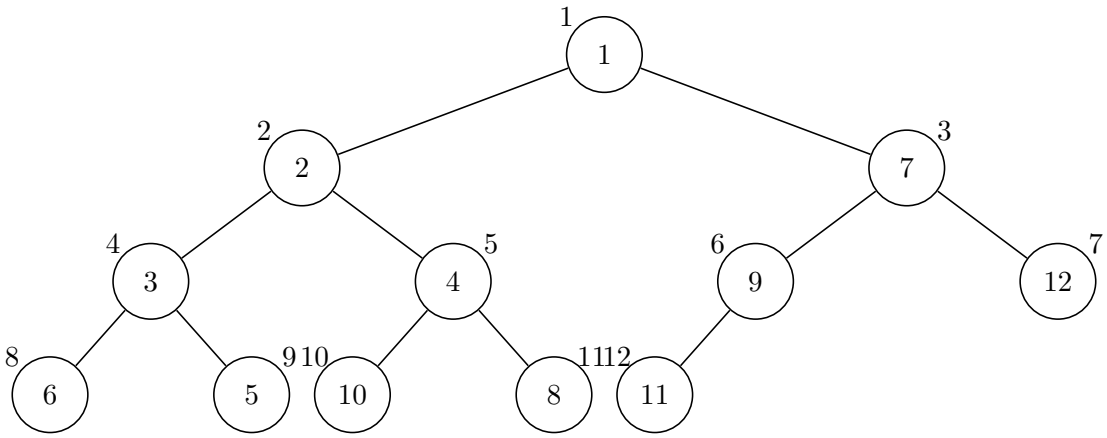
Typically the Insert, Decrease-Key, and Extract-Min operations take very small amounts of time to run—say $O(\log(n))$ time or less when the priority queue contains n elements. Creation is ideally done in linear time.

One popular implementation of priority queues is that of a *heap*. Heaps store objects according to keys just like priority queues do. It does so by maintaining a binary tree in which every vertex corresponds to an element of the heap, with the property that, if x is the parent of y in the heap, then the key of x is at most the key of y. A simple consequence of this is that the root of the tree is always an element with smallest key; this will be how we maintain the element-to-extract for the Extract-Min operation of priority queues.

An important fact about heaps is that they will not be just any binary tree. Instead, they will always be *almost complete*, in the sense that every leaf node either appears at the bottom layer, or else the layer just above the bottom. Moreover, the set of leaves in the bottom layer will always be as far left as possible. A pictorial example is given in Figure 3 of a heap containing objects with keys $1, \cdots, 12$.

One reason for this restriction is that it simplifies the implementation of heaps: if we label the root vertex as vertex 1, and recursively label the children of vertex i as $2i$ and $2i+1$, then the vertices in the tree will be labeled exactly by $1, 2, \ldots, n$, where n is the number of elements in the

Figure 3: Heap example



The vertices correspond to objects stored in the heap, with keys as represented in the diagram. Note that for every node (besides the root), its parent in the tree has a smaller key value; equivalently, every node has a key which is at most the keys of its children.

heap. This means we can actually store the heap as an *array*, which we only *conceptualize* as a tree: The children of the vertex i are just the vertices $2i$ and $2i+1$; the parent of the vertex i is just $\lfloor i/2 \rfloor$; and when we want to insert something into the heap, we just have to make a new vertex with label $n+1$. These issues are purely implementational, however, so we will still think of the heap as being a binary tree.

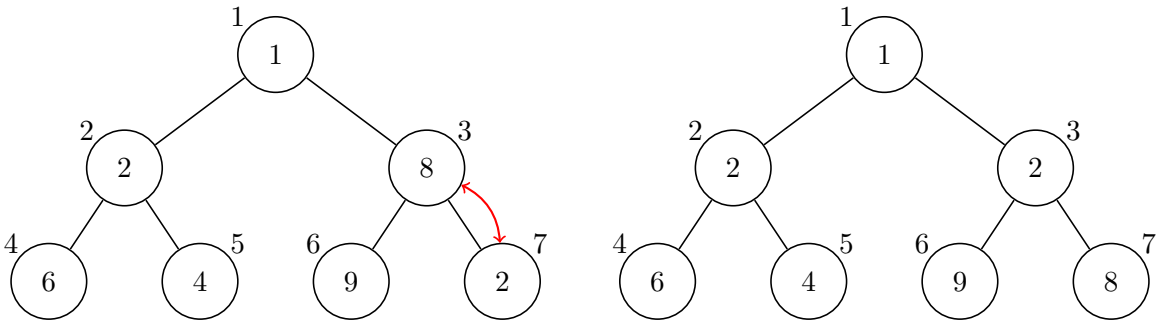
The reason that heaps make great priority queues is that we can implement the Insert, Decrease-Key, and Extract-Min operations in logarithmic time, and implement the Create operation in linear time. We will first describe the high-level approach for the Insert, Decrease-Key, and Extract-Min operations, and then discuss the linear-time creation routine.

Insert The Insert operation of a priority queue is given a new object x and a key k, and the priority queue is expected to store x according to the key k. When we implement priority queues as heaps, we can realize the Insert operation as follows: Suppose our heap currently has n elements in it. We create a new vertex v with label $n+1$, thought of as the child of $\lfloor (n+1)/2 \rfloor$. We associate to v the object x and key k.

The resulting structure is then almost a heap; however, it may be the case that the key k of v is less than the key of v's parent. In this case, we can swap v with its parent. This fixes the issue with v's parent, and does not introduce a new issue with the vertex that was v's sibling before the swap. However, it may introduce an issue between v and its new parent—but we can again address this by swapping v with its new parent. This pattern continues until eventually either v is the root of the heap, in which case no conflict can arise, or else v is not in conflict with its parent. In either case, the result is a heap. This is demonstrated in Figure 4.

When we implement the Insert operation this way, the total amount of work done is $O(\log(n))$ when the heap has n elements. This is because creating the new vertex and associating the new object and key with this vertex all takes $O(1)$ work. Each swap also takes $O(1)$ work. Moreover, since the heap is a balanced binary tree, its depth is $O(\log(n))$, and hence we perform at most $O(\log(n))$ swaps. Thus the overall time spent in the Insert operation is $O(1) + O(1) \cdot O(\log(n)) =$

Figure 4: Heap insertion example



A new object with key 2 is to be added to a heap with five elements. The new vertex is created at position seven, and associated with the inserted object and key 2. This causes a conflict in the heap between the new node and its parent (node three), since $2 < 8$, but vertex seven is below vertex three. We fix this by swapping vertices seven and three. This has the potential of introducing a conflict only between vertex three and its parent, vertex one, but this is not the case in this example; hence the result is a heap.

$O(\log(n))$.

Decrease-Key The Decrease-Key operation of a priority queue is given an element of the priority queue, and new key value k', and the priority queue is expected to update the key associated to the given element of the priority queue. For our heap implementation of priority queues, we will ignore the problem of finding the element whose key is to be updated, and assume that the Decrease-Key operation is given the vertex of the heap whose key is to be updated. In general, this is not a real problem—a program using a priority queue can store with each object its associated vertex in the heap, and we can update this value as we perform the other operations on the heap. However, it does introduce a great deal more complexity to the explanation, which is why we leave it out.[4]

The idea for implementing the Decrease-Key operation in a heap is as follows: Suppose the operation has us update the vertex v from the key k to the key k'. We start by updating the key of v to k without making any changes to the heap. Since we only ever decrease the key value ($k \leq k'$), this can only induce a conflict between v and its parent. We can address this possibility in the same way as we did for the Insert operation: if the conflict is present, we just swap v with its parent, and repeat the conflict check until either v is not in conflict with its parent, or else v is the root of the heap. One can use Figure 4 as an example of the Decrease-Key operation too: just imagine that vertex 7 had key $k' = 10$, and we were asked to update it to have key $k = 2$.

As with the Insert operation, the total work for each Decrease-Key operation is $O(\log(n))$.

[4] In fact, the C++ STL priority queue container does not implement a Decrease-Key operation due to this additional complexity. This is made more compelling by the fact that most algorithms using a priority queue with a Decrease-Key operation can be transformed into an algorithm using a priority queue without a Decrease-Key operation.

We can convey the idea of the transformation with Dijkstra's algorithm: The idea is to replace the Decrease-Key operations by Insert operations. Then the first time we extract a vertex off the priority queue, it will have the same key as if we had used Decrease-Key operations instead of Insert operations. We might extract the same vertex again later; however, we can detect this by simply keeping track of which vertices have been extracted from the priority queue (*e.g.*, with an array of booleans indexed by the vertices). The overall change in running time is that each Decrease-Key operation has been replaced by an Insert operation and (possibly) an extra Extract-Min operation. Since these are all $O(\log(n))$ operations for heap-based priority queues, the asymptotic running time doesn't change.

Extract-Min The Extract-Min operation of a priority queue is asked to remove the element with smallest key from the data structure, and return it. As noted above, the element with smallest key is easy to find in a heap: it is just the element associated to the root vertex, vertex 1. But vertex 1 is also rather difficult to remove. As it turns out, there is a simple strategy for removing it efficiently.

The basic idea is to start by making vertex 1 easier to remove: we can do this by just swapping it with vertex n. The end result is that the vertex we want to remove is now the last vertex in the heap, which means we can just remove it and still have a valid heap, except that now our heap invariant might be broken at the root. However, the heap invariant is potentially broken *only* at the root, so we can hope to employ a similar strategy as in the Insert and Decrease-Key operations.

Indeed, we will simply swap the errant vertex along the height of the tree until the heap structure is restored. The main difference is that, rather than move the errant vertex up the tree, we will instead move the errant vertex down the tree. This is more complicated than before: When going down the tree, we have to make choices; there are two children now, whereas there was only one parent before. However, if we wish to preserve the structure of the heap, there is actually only *one* child that we can swap with to fix the conflict, namely the child which has a smaller key. (If both children have the same key, then either one will work.) By repeatedly swapping the errant vertex down the tree, the heap will eventually be fixed. An example is given in Figure 5.

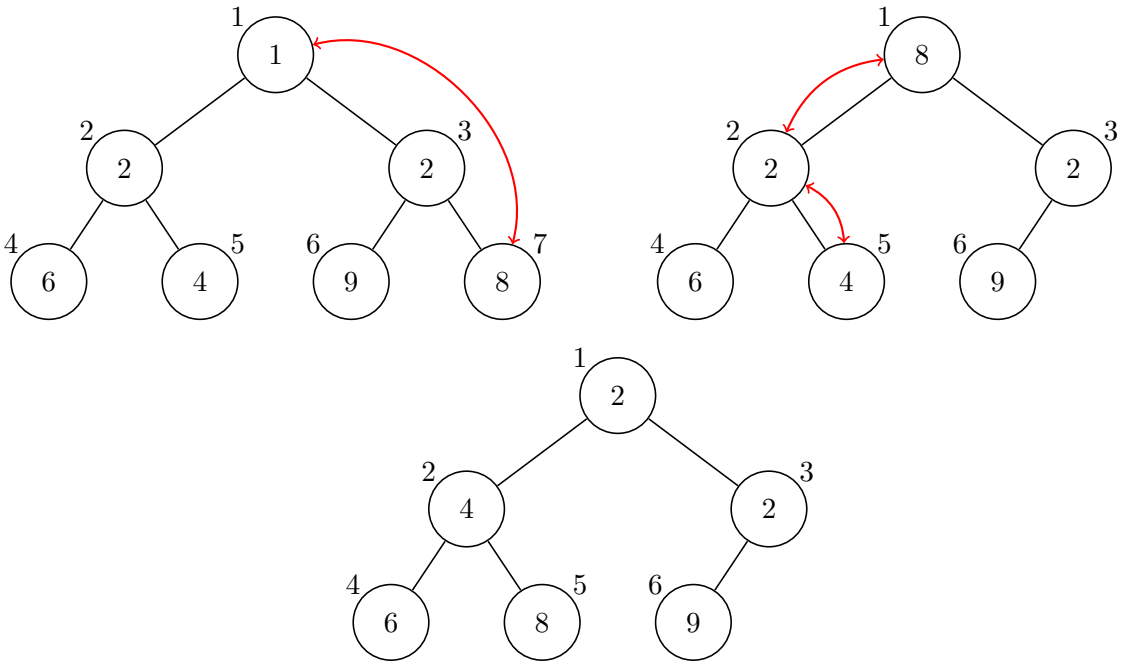
Create The Create operation for a priority queue takes as input a set of element $x_1, \ldots, x_n$ of elements with keys $k_1, \ldots, k_n$ and creates a priority queue containing exactly these elements. A naïve way to implement this for heaps is to start with an empty heap, and iteratively Insert $x_1, x_2, \ldots, x_n$. Altogether, this takes time $O(n \log(n))$. If the keys $k_1, \ldots, k_n$ are actually in order of decreasing keys, then each Insert will make the maximum number of swaps to fix the heap, and thus the creation will take $\Theta(n \log(n))$ time.

However, this is a faster way to implement the Create operation when we use heaps to implement priority queues. The basic idea is to start by placing $x_1, \ldots, x_n$ into the heap arbitrarily, ignoring the heap invariants for the time being; an easy way to do this is to put x_i at the vertex numbered i. Then we enter a phase of fixing the heap. In this case, the process is kept simple: we start from vertex n, fix the heap below vertex n, move on and fix the heap below vertex $n-1$, and so on, up until we fix the heap below vertex 1. In order to fix the heap below vertex i, note that because of the order we are iterating over the vertices, we know that the heap below vertex i satisfies the heap invariant everywhere except possibly at vertex i. Thus we can do what we did in Extract-Min: repeatedly swap the broken node its child with smallest key until the heap is fixed. The end result is a fixed heap.

A naïve bound on the running time of this procedure notes that every time we fix the heap below vertex i, we might use $O(\log(n))$ swaps to fix the heap. Thus, the whole creation uses $O(n \log(n))$ swaps to fix the heap. This isn't an improvement on repeated insertion.

However, we can improve this bound by being more careful in how we count the number of swaps when fixing vertex i. A better bound is to note that, for vertices far down in the heap, fewer swaps are needed to fix the heap. In particular, if a vertex has distance h from the leaves of the heap, then at most h swaps are needed. We know there are $O(n/2^h)$ vertices with distance h from

Figure 5: Heap Extract-Min example



An Extract-Min operation is called on the heap represented in the top left. This is handled by swapping vertices 1 and 7 to put the root of the heap as the last node in the heap. We can then remove what is now node 7, since it contains the minimum we're extracting; all that remains is to fix the conflicts between what is now node 1 and its children. Since the children of node 1 have the same key, we can pick on arbitrarily to swap with; we choose node 2. Then there is a conflict between node 2 and its children; we resolve this by swapping node 2 with node 5, since node 5 has a smaller key than node 4, the other child of node 2. After this, node 5—what was originally node 7—is now a leaf in the heap, which means there can be no more conflicts between it and its children. Thus the heap invariant has been restored.

the leaves. Thus the total amount of work over the whole creation process is given by:

$$\sum_{h=0}^{\log(n)} O(n\frac{h}{2^h})$$

where the constant hidden in the O-notation does not depend on h. Pulling the fact of n out of each term and allowing h to go to ∞, we are left with the bound

$$O(n) \cdot \sum_{h=0}^{\infty} \frac{h}{2^h}$$

The series appearing on the right converges to a constant (in fact the constant is 2). Hence the overall bound is $O(n)$.

Heap Sort We mention in passing that heap-based priority queues allow for a simple comparison-based sorting algorithm. On input an array $A[1 \cdots n]$, we put the elements of A into a priority queue, keyed on their values. Then we repeatedly Extract-Min from the priority queue until it is empty. The order of the elements obtained from the Extract-Min calls is exactly a sorted order of the elements of A.

2 Exchange Arguments

We'll now move on the examples of *exchange arguments*. The basic idea of an exchange argument is to show that any optimal solution can be transformed by some sequence of simple, small exchanges into the greedy solution. The local exchanges will preserve optimality.

A nice analogy for this kind of argument is to think of bubble sort. Suppose that the greedy solution involves outputting a sorted array (and that there are no ties in the sorting). Suppose we can also show that, in any solution, if two adjacent elements are out of order, then we can *exchange* them without making the solution worse. Then bubble sort tells us that the greedy solution has to be correct. The reason is that we can start with any optimal solution—we haven't proven it to be a sorted array yet; all we know is that it's optimal—and then bubble sort it. In bubble sort we only exchanging adjacent, out-of-order elements; this means that we always have an optimal solution after each step of bubble sort. But when bubble sort finishes, we get a sorted array, so the sorted array has to be optimal. Since greedy outputs the sorted array, it follows that greedy has to be optimal.

Note that greedy itself doesn't have to implement bubble sort—that's hardly a greedy algorithm in the first place! It might get to the solution through some other means. The point of the exchange argument is to prove the correctness of this other means by saying that it has the *same* output as a slower-but-obviously-correct algorithm.

2.1 Simple Knapsack Variant

Recall the simple knapsack problem we stated in the introduction of these notes. One proof that it works follows the exchange argument format. Recall that we're given n items with weights $w_1, w_2, \ldots, w_n$, and a capacity W on the set of items we can take. We want to take as many items as possible without exceeding the total capacity.

We gave an algorithm for this: simply sort the items in order of increasing weight, and then take every item that fits. Intuitively, the reason this works is the following: if we could fit a big item into the knapsack, then we could also fit a smaller item into it. This is already the heart of an exchange argument. Let's make it more formal.

Suppose that the items are already sorted in order of increasing weight. For the sake of notation, we'll let $G \subseteq \{1, 2, \ldots, n\}$ represent the solution that greedy outputs, and let $S \subseteq \{1, 2, \ldots, n\}$ denote any optimal solution. Then the claim of optimality is the following:

Claim 4. *$|G| = |S|$, and hence greedy yields an optimal solution.*

Proof. We know that the greedy algorithm selects an initial segment of the items—it won't refuse an item and then later take a different item. More formally, we know $G = \{1, 2, \ldots, |G|\}$. Furthermore, we know that G is the *largest* initial segment for which the items all fit into the knapsack. All we know about the optimal solution S is that it is optimal. This means it could be any set so that $|S| \geq |G|$ and so that the sum of the weights of items indexed by S is at most W.

We'll start the exchange argument with the simpler case: suppose that S is an initial segment, *i.e.*, $S = \{1, 2, \ldots, |S|\}$. Then we know that $|G| \geq |S|$, because greedy is the largest initial segment for which the items all fit into the knapsack and because the items of S fit into the knapsack. From optimality of S, we know $|S| \geq |G|$, and hence that $|G| = |S|$. (Since G and S are both initial segments, this in particular means that $G = S$, but we don't need to prove this.)

Now we'll do the more complex case: suppose that S is *not* an initial segment, *i.e.*, there exists an index i so that i is *not* in S, but $i+1$ *is* in S. We now do our exchange: Let $S' = (S - \{i+1\}) \cup \{i\}$. Then the total weight in S' is at most the total weight of the items in S, so S' indexes a set of items that fit into the knapsack. (This is because $w_i \leq w_{i+1}$.) But we also have that $|S'| = |S|$, so that S' is *no worse* of a solution than S is. Since S is optimal, it follows that S' is also optimal. It's also not too hard to see that S' is *closer to being an initial segment* than S is. That is, after repeatedly performing some finite number of these exchanges, the S' at the end is an initial segment of the items. In other words, we wind up in the simpler case above. □

2.2 Minimizing Lateness

Our next problem will be another scheduling problem, known as *minimizing lateness*. The difference with the interval scheduling problem is that we will be given the duration of jobs and a deadline by which the job should be completed, but the jobs can be started at any time. We also must run every job. Of course, there are situations in which meeting all the deadlines is impossible; in these cases, we measure the *lateness* of a job schedule by the maximum, over all jobs, of how long after the job's deadline the job finished. The objective is to find a schedule which minimizes the lateness.

Formally, the problem of minimizing lateness has the following specification:

Input: n jobs, the j-th of which is described by a duration t_j and deadline d_j.

Output: An order (or 'schedule') S in which to run the jobs. The $S(1)$-th job is run first, then the $S(2)$-th job immediately after the first finishes, and so on. Every job appears exactly once in the schedule. Every job then gets a finish time, the j-th of which is denoted by $f(S, j)$. The *lateness of a job j*, denoted $\ell(S, j)$, is the quantity $f(S, j) - d_j$ if j finishes after its deadline, or else 0 if it finishes at or before its deadline. In symbols, $\ell(S, j) = \max(0, f(S, j) - d_j)$. The *lateness of the schedule S*, denoted $\ell(S)$, is the maximum over jobs j of the lateness of

Figure 6: Minimizing Lateness, Example

(Forthcoming)

Figure 7: Minimizing Lateness, Counterexamples

(a) Smallest $d_j - t_j$ fails

(Forthcoming)

(b) Smallest t_j fails

(Forthcoming)

the j-th job. In symbols, $\ell(S) = \max_j \ell(S, j)$. The schedule S output must have *minimal* lateness over all schedules.

Let's briefly consider an example to help unravel the definitions. Figure 6 plots the two possible schedules for the input $t_1 = 1, t_2 = 3, d_1 = 4, d_2 = 3$. In the schedule $S(i) = i$ (*i.e.*, run job 1 then job 2), we have job 1 finishing at time 1 and job 2 finishing at time 4. In our notation, $f(S, 1) = 1$ and $f(S, 2) = 4$. Thus $\ell(S, 1) = 0$ and $\ell(S, 2) = 1$, and so $\ell(S) = 1$. With the other schedule, $S'(1) = 2, S'(2) = 1$, it is easy to check that $\ell(S') = 0$.

We now want to come up with a greedy strategy for minimizing lateness, *i.e.*, a simple metric by which to sort the jobs in the input. A few suggestions were given in class:

1. Sort by earliest deadline first
2. Sort by smallest value of $d_j - t_j$, *i.e.*, the latest time at which to start a job to meet its deadline
3. Sort by shortest job first

Of these, 2 and 3 are wrong, while 1 is correct. Figure 7 gives pictorial counterexamples.

To see that the first approach is correct, we'll follow the theme of this section and use an exchange argument. Let G denote the schedule produced by the greedy algorithm, and let S denote any optimal schedule. Then using the notation from the description of the problem, we can formalize the correctness of the greedy algorithm as follows:

Claim 5.

$$\ell(G) \leq \ell(S)$$

Proof. If $G = S$, then clearly $\ell(G) = \ell(S)$.

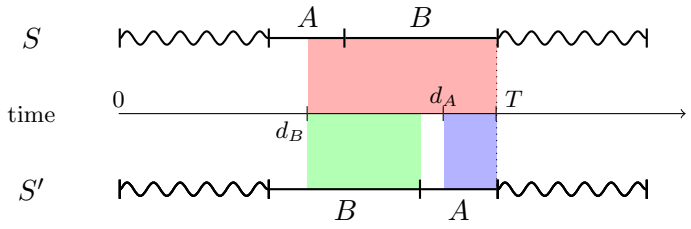
Otherwise, $G \neq S$. In particular, there must be some two *adjacent* indices i and $i + 1$ so that the jobs $S(i)$ and $S(i + 1)$ are in a different order than they appear in G. For easy of notation, let $A = S(i)$ and $B = S(i + 1)$ represent the actual jobs that are scheduled.

Consider the schedule S' which agrees with S except that we swap these two jobs. *i.e.*, $S'(j) = S(j)$ for all $j \neq i, i + 1$, and $S'(i) = B$ and $S'(i + 1) = A$. Note that this is still a valid schedule.

Suppose we can show that the lateness of S' is at most the lateness of S (*i.e.*, $\ell(S') \leq \ell(S)$). If $S' = G$, then we're done. Otherwise, note that S' is closer to G than S is, in the sense that it has fewer total inversions with respect to the greedy order. Thus, we can repeat this swapping procedure (effectively bubble-sorting S!), and ultimately wind up with G. Since at every step we did not increase the total lateness, it follows that $\ell(G) \leq \ell(S)$ for any schedule S, and thus the greedy schedule is optimal.[5] So let's now show that the lateness of S' is at most the lateness of S.

[5] If you're noticing that this argument has a very similar structure to the argument for the simple knapsack problem, then good! This pattern is the general pattern of all exchange arguments.

Figure 8: Minimizing Lateness, Proof of Greedy's Correctness



The red area is the lateness of B in the schedule S. The green and blue areas indicate the latenesses of B and A, respectively, in the schedule S'. Note that sometimes the green and blue areas may overlap (this greedy algorithm does not work when we minimize the *sum* of latenesses!)—the point is just that the blue and green areas each individually have smaller width than the red area.

Let $T = f(S, i+1)$ denote the finish time of the $(i+1)$-st job in the schedule S. Note that because we only exchange the jobs in positions i and $i+1$, the total duration of jobs in the first $i+1$ positions is the same. In other words, both S and S' finish their first $i+1$ jobs at the same time. Symbolically, we have $f(S', i+1) = T$.

Consider the effect this has on the lateness. Every job besides A and B has an identical finish time in S and in S', and thus their effect on the lateness is the same. Thus we only need to consider the effect of jobs A and B on the lateness. Since the lateness $\ell(S')$ is computed as a maximum, it suffices to show that the lateness of A within S' and the lateness of B within S' are both bounded by the lateness of S separately. In particular, we will show that both quantities are bounded by the lateness of B witin S. *i.e.*, $\ell(S', A) \leq \ell(S, B)$ and $\ell(S', B) \leq \ell(S, B)$. A pictorial representation of the general case is given in Figure 8.

One of these inequalities is straightforward. We know that job B was moved to an earlier position, so its lateness cannot increase.

For the other inequality, note that since A and B were in a different order in S than they appeared in G, we know that they appear in the same order in S' as they do in G. In other words, we know that $d_B \leq d_A$, by how the greedy order is decided. We know that job A finishes at time T in the schedule S', which is the same time that job B finishes in the schedule S. Thus we know the latenesses $\ell(S', A) = \max(0, T - d_A)$ and $\ell(S, B) = \max(0, T - d_B)$. Since $d_B \leq d_A$, it follows that $\max(0, T - d_B) \geq \max(0, T - d_A)$. (This just requires working out the cases, which we skip here.) In other words, we have $\ell(S', A) \leq \ell(S, B)$, and the proof is complete. □

2.3 Huffman Codes

(Forthcoming)

2.4 Minimum Spanning Trees

Our next problem will be that of finding minimum-weight spanning trees (henceforth: MSTs) in graphs. Formally, we have the following description:

Input: A connected, undirected graph $G = (V, E)$ on n vertices with m edges, and a weight function $w : E \to \mathbb{R}$.

Output: A spanning tree T of G for which the total weight of the edges in T is minimal.

We will actually see two algorithms for this: Kruskal's algorithm, which can be thought of as a "tree-joining" algorithm, and Prim's algorithm, which can be thought of as a "tree-growing" algorithm. We will develop intuition for each separately, but we will actually see that both can be seen to rely on the same underlying property of minimum spanning trees. This property is stated in Proposition 3, and we will prove it with an exchange argument, per the theme of this section.

Tree-joining For the tree-joining approach, we begin by understanding some of the properties that an MST must satisfy. In particular, consider a cycle in G. No MST can contain every edge in this cycle, so we might try to understand which edges are not included.

We can start with a simple case (and this will actually suffice): fix an MST T of G, and a cycle C of G, and suppose that T contains all but one edge of C. Then the missing edge must have maximal weight among the edges in C. Indeed, if T is a spanning tree containing all but one edge of a cycle C, and this edge did not have maximal weight among the edges in C, then we could remove an edge in C of maximal weight from T, and replace it with the missing edge (which has smaller weight). The result is easily seen to be a spanning tree of smaller weight than T, implying that T is not an MST.

This is actually enough to suggest the tree-joining greedy algorithm for minimum spanning trees. The algorithm is to sort the edges of G in order of increasing weight, and then greedily add edges to a tree except for when adding an edge would form a cycle in the tree. In pseudocode, we have Algorithm 3.

Algorithm 3

Input: $G = (V, E), w : E \to \mathbb{R}$
Output: A minimum spanning tree T of G

```
1: procedure MST-Kruskal((V, E), w)
2:     E_T ← ∅
3:     E_G ← Sort(E, w)                          ▷ Sort E keyed by w
4:     for e in E_G do
5:         if (V, E_T ∪ {e}) does not have a cycle then
6:             E_T ← E_T ∪ {e}
7:     return (V, E_T)
```

The output is easily seen to satisfy the property we deduced above, so we might hope that the output is actually an MST. In fact, this turns out to be the case. This can be proven directly by proving the following claim, which is a converse to our earlier observation. We will prove its correctness in a different way later.

Claim 6. *Let G, w be any connected graph with edge-weights w, and let T be a spanning tree of G. Suppose that for every cycle C of G for which T contains $|C| - 1$ edges, the edge e in C which T does not contain has maximal weight among the edges in C. Then T is a minimum spanning tree.*

Proof. (Exercise. Hint: Use an exchange argument. If you have two spanning trees of a graph, what can you say about their union? Skipping ahead to the proof we do give may also be helpful.) □

The efficiency of Algorithm 3 is not immediately clear. It's clear that Algorithm 3 has running time bounded by $O(n + m\log(m))$ plus the cost of running Line 5 for each of the m edges, but we don't know how much Line 5 contributes without knowing how exactly it is implemented. A simple DFS or BFS for each check leads to a running time on the order of $\Theta(n \cdot m)$. However, with an appropriate data structure (called a *union-find data structure*) we can bound the total contribution of Line 5 over all edges by $O(m\log(m))$. An implementation of this data structure is given following the proofs of correctness for the MST algorithms. This leads to an overall running time of $O(m\log(m))$ for Algorithm 3.

Tree-growing We now move on to the tree-growing approach to constructing minimum spanning trees. Consider the following property of minimum spanning trees:

Proposition 2. *Let $G = (V, E), w$ be any connected graph with edge-weights w, let T be a minimum spanning tree of G. Let $V' \subseteq V$ denote any nonempty subset of the vertices, and let G' and T' be the subgraphs of G and T (respectively) induced by V'. If T' is a spanning tree of G', then it is a minimum spanning tree of G'.*

(The proof is left as an exercise; see below for a hint.)

While wordy, Proposition 2 is very much analogous to Proposition 1 for shortest paths. Some property of a 'larger' object (*e.g.*, an MST or shortest path) implies the same properties of a 'smaller' object (a sub-tree or sub-shortest path). In fact, both have similar proofs: we start by supposing that the sub-object can be improved, and then replace the subobject within the larger object by the improved version to show that the larger object could also be improved. Thus the fact that the problems of finding MSTs and shortest-paths are so similar in this regard suggests that they may have similar solutions. Indeed, the 'tree-growing' algorithm is a consequence of following this intuition.

In fact, the similarities run so deep that we will actually leave out the additional intuition for the tree-growing approach. It closely resembles what was done in Section **??** for shortest paths. In fact, the pseudocode for Prim's algorithm, given in Algorithm 4, closely resembles the code for Dijkstra's algorithm (Algorithm 2. Moreover, the formal proof of correctness that we will give involves proving an appropriately modified variant of Claim 3.

The running time of Algorithm 4 is essentially identical to the analysis of Algorithm 2. The changes made for Algorithm 4 have negligible impact on the overall running time, so (with a heap-based priority queue) we get a bound of $O((m + n)\log(n))$ on the running time of Prim's algorithm.

MSTs and cuts (Here is the proposition we use for the proofs of correctness. The remaining exposition is forthcoming.)

Proposition 3. *Let (L, R) be a cut in G with L and R nonempty. Let $F \subseteq E$ be a subset of edges that do not cross the cut. Let $e^* \in E(L, R)$ be an edge of minimum weight among those edges that do cross the cut.*

If there exists a minimum spanning tree T that contains F, then there exists one that contains $F \cup \{e^*\}$.

Proof. By assumption, there exists a minimum spanning tree T of G that contains F. We need to construct a minimum spanning tree T^* of G that contains $F \cup \{e^*\}$.

Algorithm 4

Input: $G = (V, E), w : E \to \mathbb{R}$
Output: A minimum spanning tree T of G

```
1:  procedure MST-Prim(G, ℓ, s)
2:      E_T ← ∅                                          ▷ Edge set of T
3:      d(v) ← ∞ for v ∈ V                               ▷ Distance from T
4:      Nearest ← array index by V, initialized to none  ▷ Nearest neighbor in T
5:      BeenQueued ← array indexed by V, initialized to false
6:      Q ← new, empty priority queue
7:      s ← some (any) vertex in V
8:      d(s) ← 0
9:      Q.Insert(s,d(s))
10:     BeenQueued[s] ← true
11:     while Q is not empty do
12:         v ← Q.Extract-Min
13:         if v ≠ s then
14:             E_T ← E_T ∪ {{v, Nearest[v]}}            ▷ Connect v to T
15:         for u, a neighbor of v do
16:             if w({v, u}) ≥ d(u) then
17:                 continue
18:             Nearest[u] ← v
19:             d(u) ← w({v, u})
20:             if ¬BeenQueued[u] then
21:                 Q.Insert(u, d(u))
22:                 BeenQueued[u] ← true
23:             else
24:                 Q.Decrease-Key(u, d(u))
25:     return (V, E_T)
```

If T contains e^*, then we can pick $T^* = T$, and we are done.

Otherwise, adding e^* to T induces a cycle C in $T \cup \{e^*\}$. This cycle has to contain at least one edge which crosses the cut besides e^*. Let e be this edge.

Consider the subgraph T^* of G obtained by starting from T, adding e^*, and removing e. Note that:

- The number of edges of T^* is one less than the number of vertices of T^*. This is because this relationship holds for the tree T, and T^* has the same number of vertices and edges as T.
- T^* is connected. Since T is connected, it suffices to show that the two end points of $e = (u, v)$ are connected in T^*. Recall that e is part of the cycle C, and that all edges of C other than e are present in T^*. By going the long way around the cycle C from u to v (*i.e.*, not through e), we obtain a path in T^* connecting the end points of e.

The above two properties combined imply that T^* is a tree, and since T^*, like T, contains all the vertices of G, T^* is a spanning tree of G.

Moreover, since e crosses the cut and F does not contain any edge crossing the cut, we have that T^* contains all of F and thus all of $F \cup \{e^*\}$.

Finally, to argue the minimality of T^*, note that because $e \in E$ crosses the cut (L, R), we have that $w(e^*) \leq w(e)$ by how we chose e^*. Since $w(T^*) = w(T) + w(e^*) - w(e)$, this implies that $w(T^*) \leq w(T)$. Since T is a minimum spanning tree of G, it follows that so is T^* (and that $w(e) = w(e^*)$). □

Union-Find data structure (Forthcoming)