5 Greedy Algorithms

The second algorithmic strategy we are going to consider is greedy algorithms. In layman's terms, the greedy method is a simple technique: build up the solution piece by piece, picking whatever piece looks best at the time. This is not meant to be precise, and sometimes, it can take some cleverness to figure out what the greedy algorithm really is. But more often, the tricky part of using the greedy strategy is understanding whether it works! (Typically, it doesn't.)

For example, when you are faced with an NP-hard problem, you shouldn't hope to find an efficient exact algorithm, but you can hope for an approximation algorithm. Often, a simple *greedy* strategy yields a decent approximation algorithm.

5.1 Fractional Knapsack

Let's consider a relaxation of the Knapsack problem we introduced earlier. A *relaxation* of a problem is when we simplify the constraints of a problem in order to make the problem easier. Often we consider a relaxation because it produces an *approximation* of the solution to the original problem.

Exercise 5.1 (Fractional Knapsack)**.** Like the Knapsack problem, there are n items with weights $w_1, \dots, w_n$ and values $v_1, \dots, v_n$ with a Knapsack capacity of W. However, we are allowed to select a fraction of an item. The output should be a fractional subset s of the items maximizing $v(s) = \sum_i s_i v_i$ subject to the capacity constraint $\sum s_i w_i \leq W$. By a fractional subset, we mean a vector $s = (s_1, \dots, s_n)$ with all $0 \leq s_i \leq 1$.[1]

To solve this problem, let's introduce the notion of *quality* of an item: $q_i = v_i / w_i$. Intuitively, if the item was say a block of gold, it would be the dollar value of a single kg of the gold. The greedy strategy we are going to employ is going to picking items from highest to lowest quality. But first a lemma:

Lemma 5.2. *Let $q_1 > q_2$. Then in any optimal solution, either $s_1 = 1$ or $s_2 = 0$.*

Proof. A proof by contradiction. Assume an optimal solution exists with $s_1 < 1$ and $s_2 > 0$.

[1]This is the *relaxation* of the indicator vector we formulated the Knapsack problem around. In Knapsack, we restrict $s_i \in \{0, 1\}$.

Then for some small $\delta > 0$, we can define a new fractional subset by

$$s_1' = s_1 + \delta/w_1, \qquad s_2' = s_2 - \delta/w_2 \tag{5.1}$$

This will still be a fractional subset and still satisfy the capacity constraint and will increase the value by

$$\frac{v_1\delta}{w_1} - \frac{v_2\delta}{w_2} = \delta(q_1 - q_2) > 0 \tag{5.2}$$

This contradictions the assumed optimality. Therefore either $s_1 = 1$ or $s_2 = 0$. □

This tells us that if we sort the items by quality, we can greedily pick the items by best to worst quality.

Algorithm 5.3 (Fractional Knapsack)**.** Sort the items by quality so that $v_1/w_1 \geq \dots v_n/w_n$. Initialize $C = 0$ (to represent the weight of items already in the knapsack). For each $i = 1$ to n, if $w_i < W - C$, pick up all of item i. If not, pick up the fraction $(W - C)/w_i$ and halt.

Proof of Correctness. By the lemma above, we should pick up entire items of highest quality until no longer possible. Then we should pick the maximal fraction of the item of next highest quality and the lemma directly forces all other items to not be picked at all. □

Complexity. We need to only sort the values by quality and then in linear time select the items. Suppose the item values and weights are k bits integers for $k = O(\log n)$. We need to compute each q_i to sufficient accuracy; specifically, if $q_i - q_j > 0$ then

$$q_i - q_j = \frac{v_i w_j - v_j w_i}{w_i w_j} > \frac{1}{w_i w_j} \geq 2^{-2k} \tag{5.3}$$

Therefore, we only need $O(k)$ bits of accuracy. This can be computed in $\tilde{O}(k)$ time per q_i and therefore total $n\tilde{O}(\log n)$. The total sorting time for per-comparison cost of k is $O(nk \log n)$. This brings the total complexity to $O(n \log^2 n)$. □

A final note about the fractional knapsack relaxation. By relaxating the constraint to allow fractional components of items, any optimal solution to fractional knapsack $\geq$ solution to classical knapsack. Also, our solution is almost integer; only the last item chosen is fractional.

5.2 Activity Selection

Exercise 5.4 (Activity Selection)**.** Assume there are n activities each with its own start time a_i and end time b_i such that $a_i < b_i$. All these activities share a common resource (think computers trying to use the same printer). A feasible schedule of the activities is one such that no two activities are using the common resource simultaneously. Mathematically, the time intervals are disjoint: $(a_i, b_i) \cap (a_j, b_j) = \emptyset$. The goal is to find a feasible schedule that maximizes the number of activities k.

The naïve algorithm here considers all subsets of the activities for feasibility and picks the maximal one. This requires looking at 2^n subsets of activities. Let's consider some greedy metrics by which we can select items and then point out why some won't work:

1. Select the earliest-ending activity that doesn't conflict with those already selected until no more can be selected.

2. Select items by earliest start time that doesn't conflict with those already selected until no more can be selected.

3. Select items by shortest duration that doesn't conflict with those already selected until no more can be selected.

The first option is the correct one. You can come up with simple counterexamples to problems for which the second and third options don't come up with optimal feasible schedules. Choosing the right greedy metric is often the hardest part of finding a greedy algorithm. My strategy is to try to find some quick counterexamples and if I can't really think of any start trying to prove the correctness of the greedy method.

Let's prove why the first option is correct. But first a lemma:

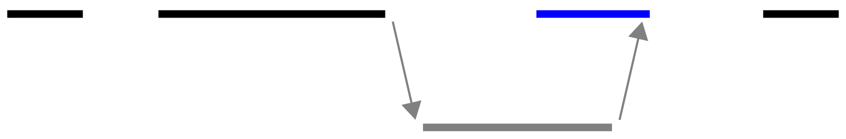

Figure 5.1: Acceptable substitution according to Lemma 5.5. The blue activity is replacable by the gray activity.

Lemma 5.5. *Suppose $S = ((a_{i_1}, b_{i_1}), \dots, (A_{i_k}, b_{i_\ell}))$ is a feasible schedule not including (a', b'). Then we can exchange in (a_{i_k}, b_{i_k}) for (a', b') if $b' \leq b_{i_k}$ and if $k > 1$, then $b_{i_{k-1}} \leq a'$.*

Proof. We are forcing that the new event ends before event k and start after event $k - 1$ See Figure 5.1. □

Theorem 5.6. *The schedule created by selecting the earliest-ending activity that doesn't conflict with those already selected is optimal and feasible.*

Proof. Feasibility follows as we select the earliest activity that doesn't conflict. Let $E = ((a_{i_1}, b_{i_1}), \dots, (a_{i_\ell}, b_{i_\ell}))$ by the output created by selecting the earliest-ending activity and $S = ((a_{j_1}, b_{j_1}), \dots, (a_{j_k}, b_{j_k}))$ any other feasible schedule.

We claim that for all $m \leq k$, (a_{i_m}, b_{i_m}) exists and $b_{i_m} \leq b_{j_m}$ (or in layman's terms the mth activity in schedule E always ends before then mth activity in schedule S). Assume the claim is false and m is the smallest counterexample. Necessarily $m = 1$ cannot be a counterexample as the schedule E by construction takes the first-ending event and must end before an event in any other schedule.

If m is a counterexample, then $b_{i_{m-1}} \leq b_{j_{m-1}} \leq a_{j_m}$ and $b_{i_m} > b_{j_m}$. This means that the event j_m ends prior to the event i_m and is feasible with the other events of E. But then event j_m would have been chosen over event i_m, a contradiction. So the claim is true.

Therefore, the number of events in E is at least that of any other feasible schedule S, proving the optimality of E. □

Let's take a moment to reflect on the general strategy employed in this proof: We considered the greedy solution E and any solution S and then proved that E is better than S by arguing that if it weren't, then it would contradict the construction of E.

5.3 Minimum Spanning Trees

Let's transition to a graph theory problem. The relevant graph definitions can be found at the beginning of Section 6 on Graph Algorithms.
All these definitions lead to the following natural question.

Exercise 5.7 (Minimum Spanning Tree)**.** Given a connected graph $G = (V, E, w)$ with positive weights, find a spanning tree of minimum weight (an MST).

Since we are interested in greedy solution, our intuition should be to build the minimum spanning tree up edge by edge until we are connected. My suggestion here is to think about how we can select the first edge. Since we are looking for minimum weight tree, let's pick the minimum weight edge in the graph (breaking ties arbitrarily). Then to pick the second edge, we could pick the next minimum weight edge. However, when we pick the third edge, we have no guarantee that the 3rd least weight edge doesn't form a triangle (i.e. 3-cycle) with the first two. So, we should consider picking the third edge as the least weight edge that doesn't form a cycle. And so on... Let's formalize this train of thought.

Definition 5.8 (Multi-cuts and Coarsenings)**.** A *multi-cut* $\mathcal{S}$ of a connected graph G is a partition of V into non-intersecting blocks $S_1, \dots, S_k$. An edge *crosses* $\mathcal{S}$ if its endpoints are in different blocks. A multi-cut *coarsens* a subgraph G' if no edge of G' crosses it.

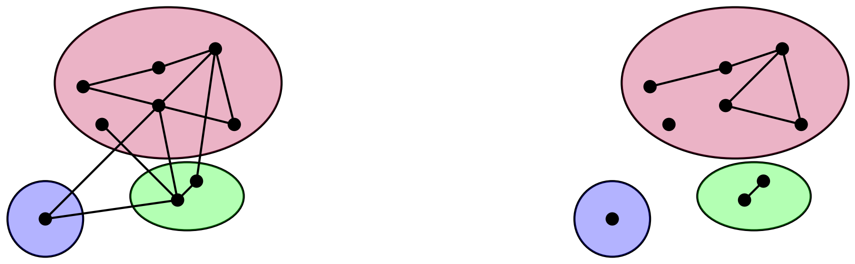
Figure 5.2: On the left, an example of a multi-cut $\mathcal{S}$ of a graph G. The partitions are the separately colored blocks. On the right, a subgraph G' of G that coarsens $\mathcal{S}$. No edge of G' crosses the partition.

We start with the empty forest i.e. all vertices and no edges: $G_0 = (V, \emptyset)$. Le the multicut $\mathcal{S}_0$ be the unique multicut where each vertex is in its unique block. Also, as there are no edges, this multicut $\mathcal{S}_0$ coarsens G_0. Our strategy will be to add the edge of minimum weight of G that crosses $\mathcal{S}_0$. This produces another forest G_1 (this time with $n-1$ trees in the forest). Let S_1 be the multicut formed by taking each tree as a block. To build G_2, we add the edge of minimum weight of G that crosses $\mathcal{S}_1$. G_2 has $n-2$ trees in the forest. We define S_2 to be the multicut formed by taking each tree as a block. And so on until we reach $\mathcal{S}_{n-1}$ which is the trivial partition. The edges in G_{n-1} form the MST.

Algorithm 5.9 (Kruskal's Algorithm for Minimum Spanning Tree)**.**

1. Sort the edges of the graph by minimum weight into a set S.
2. Create a forest F where each vertex in the graph is a separate tree.
3. While S is non-empty and F is not yet spanning
   (a) Remove the edge of minimum weight from S
   (b) If the removed edge connects two different trees then add it to F, thereby combining two trees into one.

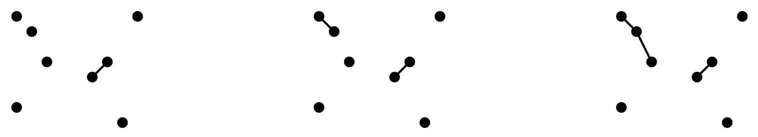
Figure 5.3: The first few iterations of Kruskal's Algorithm (Algorithm 5.9).

We have not argued the correctness of this algorithm; rather we have just explained our intuition. We will prove the correctness by generalizing this problem, and proving correctness for a whole class of greedy algorithms at once.

5.4 Matroids and Abstraction of Greedy Problems

Up till now, the examples we have seen of greedy algorithm rely on an exchange lemma. We've seen this similar property in linear algebra before.

Remark 5.10. *Given a finite-dimensional vector space X and two sets of linearly independent vectors V and W with $|V| < |W|$, there is a vector $w \in W - V$ such that $V \cup \{w\}$ is linearly independent.*

Equivalenlty, we can think of this as an exchange: If $|V| \leq |W|$ then you can remove any $v \in V$ and find a replacement from W to maintain linear independence. This notion of exchanging elements is incredibly powerful as it yields a very simple greedy algorithm to find the maximal linear independent set.

Let's consider an abstraction of greedy algorithms that will help formulate a generalized greedy algorithm.

Definition 5.11 (Matroid)**.** A matroid is a pair $(U, \mathcal{F})$, where U (the universe) is a finite set and $\mathcal{F}$ is a collection of subsets of U, satisfying

- (Non-emptyness) There is some set $I \in \mathcal{F}$ (equiv. $F \neq \emptyset$)
- (Hereditary axiom) If $I \subseteq J$ and $J \in \mathcal{F}$, then $I \in \mathcal{F}$.
- (Exchange axiom) If $I, J \in \mathcal{F}$ and $|I| > |J|$, then there is some $x \in I \setminus J$ so that $J \cup \{x\} \in \mathcal{F}$.

Definition 5.12 (Basis)**.** A basis I of a matroid $(U, \mathcal{F})$ is a maximal independent set such that $I \in \mathcal{F}$ and for any J s.t. $I \subsetneq J$ then $J \notin \mathcal{F}$.

In this context, we refer to sets in $\mathcal{F}$ as *independent sets.* A couple basic examples of matroids:

- The universe U is a finite set of vectors. We think of a set $S \subseteq U$ as independent if the vectors in S are linearly independent. A basis for this matroid is a basis (in the linear algebra sense) for the vector space spanned by U.
- The universe U is again a finite set of vectors. But this time, we think of a set $S \subseteq U$ as independent if $\text{Span}(U \setminus S) = \text{Span}(U)$. (This is the *dual matroid* to the previous matroid.) A basis for this matroid is a collection of vectors whose *complement* is a basis (in the linear algebra sense) for $\text{Span}(U)$.
- Let $G = (V, E)$ be a connected undirected graph. Then the universe U is the set of edges E and $\mathcal{F} =$ all acyclic subgraphs of G (forests).
- Let $G = (V, E)$ be a connected undirected graph again. Again $U = E$ but $\mathbf{F} =$ the subsets of E whose complements are connected in G. (This is the dual matroid of the previous example.)

Definition 5.13 (Weighted Matroid)**.** A *weighted matroid* is a matroid $(U, \mathcal{F})$ together with a weight function $w : U \to \mathbb{R}^+$. We may sometimes extend the function w to $w : \mathcal{F} \to \mathbb{R}^+$ by $w(A) = \sum_{u \in A} w(u)$ for any $A \in \mathcal{F}$.

Note: We assume that the weight of all elements is positive. If not, we can simply ignore all the items of negative weight initially as it is necessarily disadvantageous to have them in the following problem:

In the maximum-weight matroid basis problem, we are given a weighted matroid, and we are asked for a basis B which maximizes $w(B) = \sum_{u \in B} w(u)$. For the maximizes-weight matroid basis problem, the following greedy algorithm works:

Algorithm 5.14 (Matroid Greedy Algorithm)**.**

1. Initialize $A = \emptyset$.
2. Sort the elements of U by weight.
3. Repeatedly add to B the maximum-weight point $u \in U$ such that $A \cup \{u\}$ is still independent (i.e. $A \cup \{u\} \in \mathcal{F}$), until no such u exists.
4. Return A.

Let's prove the correctness of the remarkably simple matroid greedy algorithm.

Lemma 5.15 (Greedy choice property)**.** *Suppose that $M = (U, \mathcal{F})$ is a weighted matrioid with weight function $w : \mathcal{U} \to \mathbb{R}$ and that U is sorted into monotonically decreasing order by weight. Let x be the first element of U such that $\{x\} \in \mathcal{F}$ (i.e. independent), if any such x exists. If it does, then there exists an optimal subset A of U containing x.*

Proof. If no such x exists, then the only independent subset is the empty set (i.e. $\mathcal{F} = \{\emptyset\}$) and the lemma is trivial. Assume then that $\mathcal{F}$ contains some non-empty optimal subset B. There are two cases: $x \in B$ or $x \notin B$. In the first, taking $A = B$ proves the lemma. So assume $x \notin B$.

Assume there exists $y \in B$ such that $w(y) > w(x)$. As $y \in B$ and $B \in \mathcal{F}$ then $\{y\} \in \mathcal{F}$. If $w(y) > w(x)$, then y would be the first element of U, contradicting the construction.

Therefore, by construction, $\forall\ y \in B$, $w(x) \geq w(y)$.

We now construct a set $A \in \mathcal{F}$ such that $x \in A, |A| = |B|$, and $w(A) \geq w(B)$. Applying the exchange axiom, we find an element of B to add to A while still preserving independence. We can repeat this property until $|A| = |B|$. Then, by construction $A = B - \{y\} \cup \{x\}$ for some $y \in B$. Then as $w(y) \leq w(x)$,

$$w(A) = w(B) - w(y) + w(x) \geq w(B) \tag{5.4}$$

As B is optimal, then A containing x is also optimal. □

Lemma 5.16. *If $M = (U, \mathcal{F})$ is a matroid and x is an element of U such that there exists a set A with $A \cup \{x\} \in \mathcal{F}$, then $\{x\} \in \mathcal{F}$.*

Proof. This is trivial by the hereditary axiom as $\{x\} \subseteq A \cup \{x\}$. □

Lemma 5.17 (Optimal-substructure property)**.** *Let x be the first element of U chosen by the greedy algorithm above for weighted matroid $M = (U, \mathcal{F})$. The remaining problem of finding a maximum-weight independent subset containing x reduces to finding a maximum-weight independent subset on the following matroid $M' = (U', \mathcal{F}')$ with weight function w' defined as:*

$$U' = \{y \in U \mid \{x, y\} \in \mathcal{F}\}, \qquad \mathcal{F}' = \{B \subseteq U - \{x\} \mid B \cup \{x\} \in \mathcal{F}\}, \qquad w' = w|_{U'} \tag{5.5}$$

M' is called the contraction of M.

Proof. If A is an optimal independent subset of M containing x, then $A' = A - \{x\}$ is an independent set of M'. Conversely, if A' is an independent subset of M, then $A = A \cup \{x\}$ is an independent set of M. In both cases $w(A) = w(A') + w(x)$, so a maximal solution of one yields a maximal solution of the other. □

Theorem 5.18 (Correctness of the greedy matroid algorithm)**.** *The greedy algorithm presented in Algorithm 5.14 generates an optimal subset.*

Proof. By the contrapositive of Lemma 5.16, if we passover choosing some element x, we will not need to reconsider it. This proves that our linear search through the sorted elements of U is sufficient; we don't need to loop over elements a second time. When the algorithm selects an initial element x, Lemma 5.15 guarantees that there is some optimal independent set containing x. Finally, Lemma 5.17 demonstrates that we can reduce to finding an optimal independent set on the contraction of M is sufficient. Its easy to see that Algorithm 5.14 does precisely this, completing the proof. $\square$

Even though that was a long proof for such a simple statement, it has given us an incredible ability to demonstrate the correctness of a greedy algorithm. All we have to do is express a problem in the matroid formulation and presto! we have an optimal algorithm for it.

Theorem 5.19 (Runtime of the Greedy Matroid Algorithm)**.** *The runtime of Algorithm 5.14 is $O(n \log n + nf(n))$ where $f(m)$ is the time it takes to check if $A \cup \{x\} \in \mathcal{F}$ is independent given $A \in \mathcal{F}$ with $|A| = m$.*

Proof. The sorting takes $O(n \log n)$ time[2] followed by seeing if the addition of every element of U to the being built optimal independent set maintains independence. This takes an addition $O(nf(n))$ time, proving the runtime. $\square$

5.5 Minimum Spanning Tree

5.5.1 Kruskal's Algorithm

We introduced Kruskal's Algorithm already as Algorithm 5.9 but we never proved its correctness or argued its runtime. We can prove the algorithms correctness by phrasing the algorithm as a matroid. In this case the universe $U = E$ and $\mathcal{F} =$ the set of all forests in G. Convince yourself that $(U, \mathcal{F})$ is indeed a matroid.

Corollary 5.20. *The Minimum Spanning Tree problem can be solved in $O(n^2 + m \log n)$ runtime where $n = |V|$ and $m = |E|$.*

Proof. Correctness then followed as an immediate consequence of formulation as a matroid and furthermore we got an initial bound on the runtime. Sorting the edges takes

[2]Note that this assumes that calculating the weight of any element x is $O(1)$. In reality, this time should also be taken into account.

$O(m \log m) = O(m \log n)$ time.[3] The time it takes to check if adding an edge to a pre-existing forest generates a new forest is naively $O(n)$ if we keep track of the vertices in each tree of the forest. Then if we are edge an edge (v_1, v_2) we only need to check if v_2 is in the tree that v_1 is. If it isn't, then we can add the edge and we adjust our records accordingly to indicate that the two trees were merged. Thus the total time required for this part of the algorithm is $O(n^2)$. Thus, the total runtime is $O(n^2 + m \log n)$. □

However, we can do better by using a different data structure to keep track of the forest so far. Notice, that if the graph is dense meaning that $m = \Omega(n^2)$, then this algorithm is optimal. However if $m = o(n^2)$, then we can do better if we can reduce the $O(n^2)$ term to $O(m \log n)$. Specifically, in order to bring the runtime down to $O(m \log n)$ we want to be able to perform $O(n)$ checks of adding an edge to a pre-existing forest generates a forest in $O(m \log n)$ time. The data structure we are going to use is a *disjoint-set data structure*. We actually will do the checks in total $O(n \log n)$ time.[4]

5.5.2 Disjoint-Set Data Structure

Also known as a union-find data structure, this data structure is optimized to keep track of elements partitioned into a number of disjoint subsets. The structure is generally designed to support two operations:

- *Find*: Determine which subset an element is in. Note, testing if two elements are in the same subset can be achieved by two find calls.
- *Union*: Join two subsets together.

There are a lot of ways that we could create a disjoint-set data structure. The simplest way is each set is stored as a list. Then find would take $O(n)$ as we would have to search every list. However, union would be $O(1)$ by simplying pointing the tail of one list to the head of the other. This is the structure, I alluded to in Corollary 5.20 and clearly would not solve the problem we have.

The better approach is to utilize the existing tree structure. We can define each tree in the forest as its own disjoint subset. Furthermore, as a subset is a tree, we can identify each

[3] As $m \leq n^2$, then $\log m \leq 2 \log n$ so $O(\log m) = O(\log n)$.
[4] Also $m \geq n - 1 = O(n)$ for connected graphs.

subset with the root of the tree.

Assume that we kept for each vertex x, we kept track of its parent $\pi(x)$ in the tree and set the parent of the root to itself. Then find(x) is simply recursively calling π until we get to r such that $\pi(r) = r$. The runtime of this is the depth of the tree.

To support the union operation is also simple. To run union(x, y) calculate roots r_x and r_y using find and then set $\pi(r_x) \leftarrow r_y$. However, this will run into issues that in worst case the tree is going to be severly unbalanced. To reconcile this, keep track of with each root, the depth of the tree. Point the root of the tree with smaller depth to the new tree. The depth of the new tree is at most 1 + the greater depth.

With a little more work, you can argue that the trees are going to be fairly balanced. In which case, the depth of any tree is at most $O(\log n)$. So find runs in $O(\log n)$. Since union requires running two find operations and then some $O(1)$ adjustments, its runtime is also $O(\log n)$.

Returning to Kruskal's algorithm, we can apply this data structue to bring the total runtime down to $O(n \log n + m \log n) = O(m \log n)$. We will soon beat this with Prim's algorithm.

Theorem 5.21. *Kruskal's algorithm for Minimum Spanning Tree has a faster runtime of $O(m \log n)$.*

5.5.3 Metric Steiner Tree Problem

Exercise 5.22 (Metric Steiner Tree Problem)**.** Given a finite metric space $G = (V, E, w)$, i.e. $w : E \to \mathbb{R}^+$ satisfies the triangle inequality), and a subset $R \subset V$ of the vertices, find the minimum-weight tree that contains R.

Although this problem looks similar to the MST problem, it is in fact significantly harder. It's actually NP-hard. But, it does exhibit a 2-factor poly. greedy algorithm.

Algorithm 5.23 (Metric Steiner Tree Greedy Algorithm)**.** We compute the all-pair shortest path function $w' : \binom{R}{2} \to \mathbb{R}^+$. We will discus how to do this later. $w'(r_1, r_2)$ is the length of the shortest path $r_1 \rightsquigarrow r_2$. Then, we compute the MST on R under metric w'.

Proof of Correctness. We need to show that this is a 2-factor algorithm. We argue that given a Steiner tree S, that $\exists$ a spanning path T in (R, w') that depth-first searches S, using each of its edges at most twice.

TODO: INSERT IMAGES FROM SCHULMAN SLIDES.

5.5.4 Prim's Algorithm

An alternative algorithm to Kruskal's that performs slightly faster is Prim's algorithm. In Prim's, instead of starting with the n trees consisting of single vertices and connecting trees to form a single tree, we start with an arbitrary vertex and grow the tree by adding the greedy edge. By the greedy edge, I mean the edge of least weight connecting from the tree to vertices not yet connected.

Algorithm 5.24 (Prim's)**.** Given a graph with a weight function $G = (V, E, w)$:

- Make a Priority Queue Q and add all vertices v to Q at distance ∞.
- Choose a root $r \in V$ arbitrarily. Decrease the key of r in Q to 0.
- While Q is non-empty
  - Set $u \leftarrow$ DeleteMin(Q).
  - For all edges $(u, v) \in E$, decrease the key of v in Q to $\min(\text{key}(Q, v), w(u, v))$.z

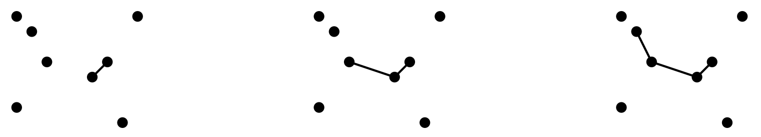

Figure 5.4: The first few iterations of Prim's Algorithm (Algorithm 5.24).

There are n inserts to the Q, n min-deletions, m decrease-keys called. If we implement it with a binary heap, then each of the operations is a $O(\log n)$ operation and the total complexity is $O(n \log n + m \log n) = O(m \log n)$.

However, if we build it witha Fibonacci, then both insertion and decrease-key are $O(1)$ and min-deletions are $O(\log n)$ giving the runtime of $O(m + n \log n)$.[5]

[5]We haven't discussed the implementation of the Fibonacci heap and we won't in this class. But this is

5.6 Clustering and Packing

An important problem, especially in Machine Learning, is the problem of clustering. Assume we have a metric space (V, w) where $w : V^2 \to \mathbb{R}^+$ is a metric i.e. $w(x, y) = 0$ iff $x = y$, $w(x, y) = w(y, x)$ and w obeys the triangle inequality. Our goal is to partition V into k blocks such that points in the same block are 'nearby' and points in different blocks are 'far' apart. This is not meant to be formal.

Most formalizations of 'near' and 'far' result in NP-hard optimization problems (in particular they are non-convex). Fortunately, most of these problems exhibit simple greedy strategies. However, the first problem we will see can actually be solved optimally through a greedy algorithm.

5.6.1 Maxi-min Spacing

Definition 5.25 (Multi-cut)**.** A multi-cut $\mathcal{S}$ is a set of subsets $\{S_1, \ldots, S_k\}$ of V such that $V = S_1 \cup \ldots \cup S_k$ and $S_i \cap S_j = \emptyset$ for all $1 \leq i < j \leq k$.

Definition 5.26 (Spacing)**.** The spacing of a cut $\mathcal{S}$, denoted spacing($\mathcal{S}$) is the least weight edge crossing $\mathcal{S}$.

$$\text{spacing}(\mathcal{S}) = \min_{u \in S_i, v \in S_j, i \neq j} w_{u,v} \tag{5.6}$$

Exercise 5.27 (Maxi-min Spacing Clustering Problem)**.** Given (V, w, k) find a k-way cut $\mathcal{S}$ with maximal spacing.

Lemma 5.28. *Running Kruskal's Algorithm until there are only k trees left produces a maxi-min spacing clustering.*

Proof. Let $\mathcal{S}$ be the Kruskal cut and $\mathcal{R}$ any other cut. Then as $\mathcal{S}$ and $\mathcal{R}$ both cover V, $\exists i$ s.t. S_i intersects two blocks of $\mathcal{R}$. Wlog, call these blocks R_1 and R_2 and as their intersections with S_i are non-empty, exist elements $a \in S_i \cap R_1$ and $d \in S_i \cap R_2$.

Consider the path $a \rightsquigarrow d$ in S_i. As $a \in R_1$ and $d \in R_2$ then there is an edge $b \sim c$ in $a \rightsquigarrow d$ crossing $\mathcal{R}$. But, recall how the Kruskal forest is built; namely edges are added in

a useful result to know. Fibonacci heaps have a better amortized runtime than most other priority queue data structures. Their name comes from their analysis which involves Fibonacci numbers. While they do have an amortized runtime that is faster, they are very complicated to code. Furthermore, in practice they have been shown to be slower because they involve more storage and manipulations of nodes.

non-decreasing order of weight. Therefore $w(b,c) \leq \text{spacing}(S)$. However, as $b \sim c$ crosses $\mathcal{R}$, $\text{spacing}(\mathcal{R}) \leq w(b,c)$. Therefore,

$$\text{spacing}(\mathcal{R}) \leq w(b,c) \leq \text{spacing}(\mathcal{S}) \tag{5.7}$$

Thus $\mathcal{S}$ is optimal to $\mathcal{R}$. As $\mathcal{R}$ was chosen arbitrarily, $\mathcal{S}$ is optimal. □

It should be noted that this algorithm gives us something more than just the maxi-min spacing clustering. The structure of each of the k trees gives us an MST for each cluster.

5.6.2 k-Center Covering and Packing

Definition 5.29 (Ball)**.** Let $G = (V, w)$ be a finite metric space. The (closed) ball with center x and radius r denoted $B(x,r)$ is

$$B(x,r) = \{v \in V : w(x,v) \leq r\} \tag{5.8}$$

The open ball $B^{\circ}(x,r)$ is the same definition except with a strict inequality $<$ instead of $\leq$.

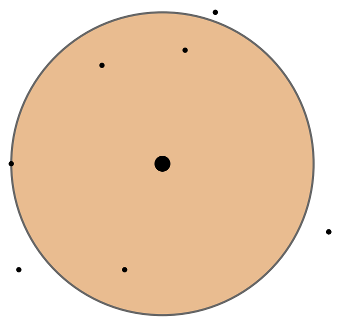

Figure 5.5: The points in orange are in the closed and open ball. The points on the boundary are only in the closed ball. The points outside are in neither.

Definition 5.30 (Covering)**.** Given points $x_1, \ldots, x_k \in V$, and a radius r, we say that $B(x_1,r), \ldots, B(x_k,r)$ are a (r,k)-covering of V if

$$V \subseteq \bigcup_{i=1}^{k} B(x_i, k) \tag{5.9}$$

The obvious two questions to ask for any finite metric space G are:

- For a fixed radius r, what is the *minimal* number k of balls needed to cover V?[6] We will notate this problem as $K_{\text{cover}}(V, r)$ or $K_{\text{cover}}(r)$ when the metric space is obvious.
- For a fixed number k of balls, what is the *minimal* radius r such that V has a (r, k) cover? We notate this problem as $R_{\text{cover}}(V, k)$ or $R_{\text{cover}}(k)$.

Theorem 5.31 (Hsu & Nemhauser '79)**.** *It is NP-hard to approximate $R_{\text{cover}}(k)$ to any multiplicative-factor better than 2.*

Theorem 5.32 (Feder & Greene '88)**.** *If the points belong to Euclidean-space, then it is NP-hard to approximate $R_{\text{cover}}(k)$ to any multiplicative-factor better than 1.82.*

Let's however see a 2-factor greedy approximation algorithm for $R_{\text{cover}}(k)$. First a definition.

Definition 5.33 (Distance to a Set)**.** Given a finite metric space $G = (V, w)$, a point $x \in V$, and a non-empty subset $S \subseteq V$, the distance $w(x, S)$ is defined by

$$w(x, S) = \min_{y \in S} w(x, y). \tag{5.10}$$

Algorithm 5.34 (Gonzalez '85, Hockbaum-Shmoys '86 for k-cover)**.** Pick any $x_1 \in V$. Then for $i = 2, \ldots, k$ (in order), choose x_i to be the furthest point from $x_1, \ldots, x_i$.

Proof of Correctness. For notational convenience define

$$R_C(x_1, \ldots, x_j) = \min \; r \text{ s.t. } \left(V \subseteq \bigcup_{i=1}^{j} B(x_i, r) \right) \tag{5.11}$$

By definition, for optimal $x_1^\star, \ldots, x_k^\star$, $R_{\text{cover}}(k) = R_C(x_1^\star, \ldots, x_k^\star)$. Notice equivalently that

$$R_C(x_1, \ldots, x_j) = \max_{v \in V} w(v, \{x_1, \ldots, x_j\}) \tag{5.12}$$

This is because every point $v \in V$ must be covered and in order to be covered the radius must be at least the distance from v to $\{x_1, \ldots, x_j\}$. The maximum distance overall v sufficiently covers all of V. Any smaller and some $v \in V$ isn't covered.

[6]Note, we provide no restriction of where the centers of the balls must be.

Let $x_1, \ldots, x_k$ be the points as picked by the algorithm. Notice that $w(x, S) \geq w(x, S \cup \{y\})$. Therefore by (5.12), adding a point x_j to the set of centers will only decrease the minimum cover radius. See Figure 5.6 for an illustration. Therefore,

$$R_C(x_1) \geq \ldots \geq R_C(x_1, x_2, \ldots, x_j) \geq \ldots \geq R_C(x_1, \ldots, x_n) = 0 \tag{5.13}$$

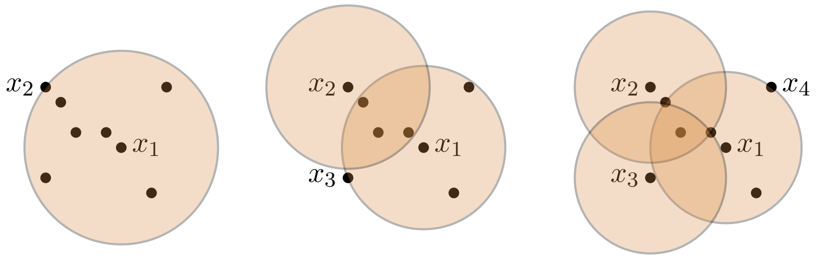

Figure 5.6: Progression of the greedy algorithm for $R_{\text{cover}}(k)$. x_1 is chosen arbitrarily and x_2, x_3, x_4 are chosen as the furthest point from the previously chosen points.

Consider how a point x_{k+1} would be selected. It would be selected as the furthest point from $x_1, \ldots, x_k$. Therefore,

$$w(x_i, x_{k+1}) \geq R_C(x_1, \ldots, x_k) \tag{5.14}$$

Adding to that (5.13), we get that the distance between any two points of $x_1, \ldots, x_k$ is at least $R_C(x_1, \ldots, x_k)$.

Notice that no ball of radius $r < R_C(x_1, \ldots, x_k)/2$ can cover two points of $x_1, \ldots, x_{k+1}$. This is a triangle inequality argument. Therefore, no choice of k centers for balls of radius $r < R_C(x_1, \ldots, x_k)/2$ will cover all $k+1$ points.

Therefore the minimal covering radius is at least $R_C(x_1, \ldots, x_k)/2 = R_{\text{cover}}(k)/2$, completing the proof. □

The other set of problems we are interested in are packing problems.

Definition 5.35 (Packing)**.** Given a finite metric space $G = (V, w)$, a (k, r)-packing is a set of pairwise disjoint balls $B(x_1, r), \ldots, B(x_k, r)$ such that $x_1, \ldots, x_k \in V$. By pairwise disjoint we mean for any $1 \leq i < j \leq k$, $B(x_i, r) \cap B(x_j, r) = \emptyset$.[7]

The analagous two questions can be asked for packings. Notice that the maximization and minimizations have been switched however:

- For a fixed radius r, what is the *maximal* number k of balls capable of being packed in V?[8] We will notate this problem as $K_{\text{pack}}(V, k)$ or $K_{\text{pack}}(r)$ when the metric space is obvious.
- For a fixed number k of balls, what is the *maximal* radius r such that V has a (r, k) packing? We notate this problem as $R_{\text{pack}}(V, k)$ or $R_{\text{pack}}(k)$.

These two problems, covering and packing are in fact duals of one another. Meaning that the optimal solution of one gives a bound on the optimal solution of the other.

Theorem 5.36 (Covering and Packing Weak Duality)**.** *$K_{\text{pack}}(r) \leq K_{\text{cover}}(r)$.*

Proof. Assume for contradiction that duality does not hold. Then $K_{\text{pack}}(r) > K_{\text{cover}}(r)$. By the pidgeon-hole principle, at least two packing centers y_1, y_2 are contained in one of the covering balls $B(x, r)$. But then $x \in B(y_1, r)$ and $x \in B(y_2, r)$ so the balls are not pairwise disjoint. This contradicts being a packing. $\square$

We leave the following other weak duality as an exercise:

Theorem 5.37 (Covering and Packing Weak Duality)**.** *$R_{\text{pack}}(k + 1) \leq R_{\text{cover}}(k)$.*

We can sumarize all these results into this neat table:

Find	Cover	Pack	Weak Duality
K	$\min\{k : \exists(k, r) \text{ covering}\}$	$\max\{k : \exists(k, r) \text{ packing}\}$	$K_{\text{pack}}(r) \leq K_{\text{cover}}(r)$
R	$\inf\{r : \exists(k, r) \text{ covering}\}$	$\sup\{r : \exists(k, r) \text{ packing}\}$	$R_{\text{pack}}(k + 1) \leq R_{\text{cover}}(k)$

[7] A word of warning: It is tempting to draw the balls for a packing in Euclidean geometry when trying to develop an intuition for this idea. Even if two balls overlap when drawn on paper, we are only interested in their overlap in regards to the finite metric space. Meaning, that the balls can overlap when drawn if there are no points of V in the overlapping region.

[8] Note, we provide no restriction of where the centers of the balls must be.