COMP 251 - Greedy Algorithms

Algorithms and Data Structures

Written By

David Knapik

McGill University Department of Mathematics and Statistics
david.knapik@mcgill.ca

February 27, 2018

Contents

1

Introduction - Scheduling

Greedy algorithms make locally optimal (**myopic**) choices at each step. They are fast, simple, and easy to code BUT usually are trash.

1.1 Task Scheduling

Suppose a firm receives job orders from n customers. This firm can only do one task at a time. Let the time it takes to complete the job of customer i be denoted t_i. Each customer wants their job done as fast as possible. Assume the firm processes the jobs in the order $\{1, 2, ..., n\}$ and that the wait time of customer l is

$$w_l = \sum_{i=1}^{l} t_i$$

To be capitalist and maximize the profits, the objective of the firm is to minimize the sum of the waiting times

$$\sum_{l=1}^{n} w_l = \sum_{l=1}^{n} \sum_{i=1}^{l} t_i$$

Algorithm 1.1.1 (Greedy Task Scheduling Algorithm)**.**

1. Sort the jobs by length $t_1 \leq t_2 \leq \cdots \leq t_n$.
2. Schedule jobs in the order $\{1, 2, ..., n\}$.

Theorem 1.1.1. *The Greedy Task Scheduling algorithm 1.1.1 works and is optimal.*

Proof. A commonly used way to prove that Greedy Algorithms work is by using an exchange argument. We shall do so here.

Let the greedy algorithm schedule in the order $\{1, 2, ..., n\}$. Assume $\exists$ a better schedule $\mathcal{S}$. Then, there is a pair of jobs i, j such that

1. Job i is scheduled immediately before job j by $\mathcal{S}$
2. Job i is longer than job j: $t_i > t_j$

But then, exchanging the order of jobs i, j gives a better schedule $\hat{\mathcal{S}}$. Observe that the waiting time of the other jobs remains the same i.e. $\hat{w_k} = w_k \ \forall k \neq i, j$. But, clearly $\hat{w_i} + \hat{w_j} < w_i + w_j$. This contradicts that $\mathcal{S}$ was optimal.

□

1.1.1 Run-time

We just needed to sort n time lengths. Thus, the run-time is $O(n \log n)$. So, the Greedy Task Scheduling algorithm 1.1.1 is efficient.

1.2 Class Scheduling (an Interval Selection Problem)

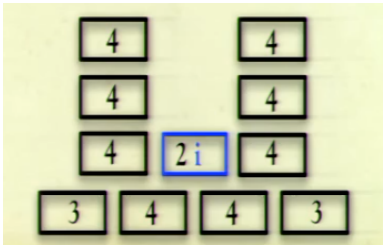
Suppose there is a set $I = \{1, ..., n\}$ of classes that request to use a room. Class i has start time s_i and finish time f_i. We want to book as many classes into the room as possible. However, we cannot book 2 classes that need to use the room at exactly the same time.

What kind of of greedy algorithm would we use for this? Here are some options:

1. First-Start: we select the class that starts earliest and iterate on the remaining classes that do not conflict with this selection. This algorithm is bad and doesn't work. For example, the first class could be all day long

2. Shortest duration: we select the class of shortest duration, and iterate on the remaining classes that do not conflict with this selection. This algorithm doesn't work. For example we could have a bad ordering and thus many clashes.

3. Minimum-Conflict: we select the class that conflicts with the fewest number of classes and then iterate. This algorithm doesn't work.



4. Last-Start: we select the class that starts last and iterate.

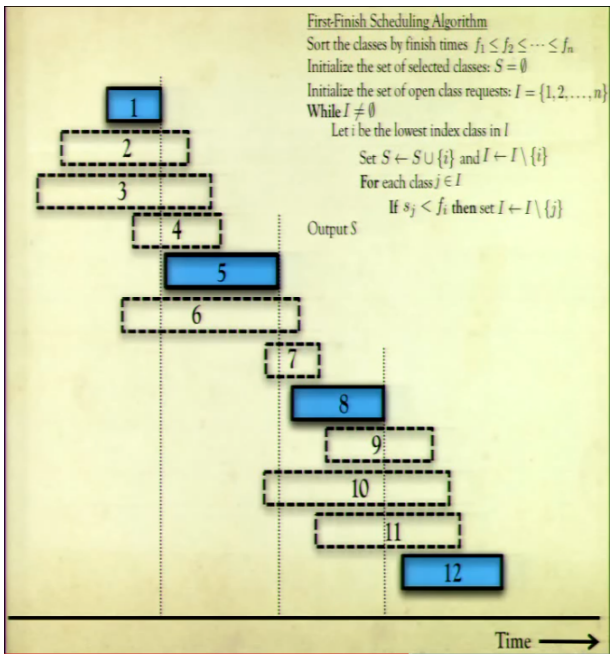
Last-Start actually works and outputs an optimal schedule. It is symmetric to First-Finish where we select the class that finishes first and iterate on the classes that do not conflict with this selection.

Algorithm 1.2.1 (FirstFinish(I))**.**

1. Let Class 1 be the class with the earliest finish time.

2. Let X be the set of classes that clash with Class 1.

3. Output $\{1\} \cup$ FirstFinish $(I \setminus X)$.



Theorem 1.2.1. *The FirstFinish scheduling algorithm 1.2.1 outputs an optimal schedule.*

Before we prove this, we present the following lemma:

Lemma 1.2.1. There is an optimal schedule that selects Class 1.

Proof. Take an optimal schedule $\mathcal{S}$ with $1 \notin \mathcal{S}$. Let i denote the lowest index class selected in $\mathcal{S}$.

We claim that $(\mathcal{S} \setminus \{i\}) \cup \{i\}$ is a feasible allocation of max. size.

Indeed this follows since: $f_i \leq s_j$ for any $j \in \mathcal{S} \setminus \{i\}$. But $f_1 \leq f_i \leq s_j$ so Class 1 doesn't conflict with any class in $\mathcal{S} \setminus \{i\}$. Thus, $(\mathcal{S} \setminus \{i\}) \cup \{i\}$ is feasible with cardinality equal to $|\mathcal{S}|$.

□

Now, back to the proof of the theorem:

Proof. We proceed via induction on $|opt(I)|$. Our base case is that $|opt(I)| = 1$ and clearly FirstFinish outputs $\{1\}$ so we are done. Now our hypothesis is that if $|opt(I)| = k$ then FirstFinish gives an optimal solution. Let $|opt(I)| = k + 1$.

FirstFinish outputs $\{1\} \cup$ FirstFinish$(I \setminus X)$. By our lemma, $\{1\} \in \mathcal{S}^*$ for some optimal solution $\mathcal{S}^*$. Thus $\mathcal{S}^* \setminus \{1\}$ is an optimal solution for the sub-problem $I \setminus X$. So $|opt(I \setminus X)| = k$. And by the induction hypothesis, FirstFinish gives an optimal solution for the sub-problem $I \setminus X$. $\rightarrow |FirstFinish(I \setminus X)| = k$ and thus

$$|\{1\} \cup FirstFinish(I \setminus X)| = k + 1$$

□

1.2.1 Run-time

There are at most n iterations and it takes $O(n)$ time to find the class that finishes earliest in each iteration. Thus, the run-time is $O(n^2)$. But, this was a very crude analysis. With a more subtle implementation and analysis, we can actually get $O(n \log n)$ (exercise!).

2

The Shortest Path Problem

Say that we have a directed graph $G = (V, A)$ with a source vertex $s \in V$. Let each arc $a \in A$ have non-negative length l_a. The length of a path P in the graph is then $l(P) = \sum_{a \in P} l_a$. The problem we consider here is that we want to find the shortest length paths from s to every other vertex $v \in V$. Amazingly, we can actually do this in 1 go, rather than running an algorithm say n times. We do this via the greedy algorithm know as Dijkstra's Shortest Path Algorithm which was conceived by the Dutch computer scientist Edsger W. Dijkstra in 1956.

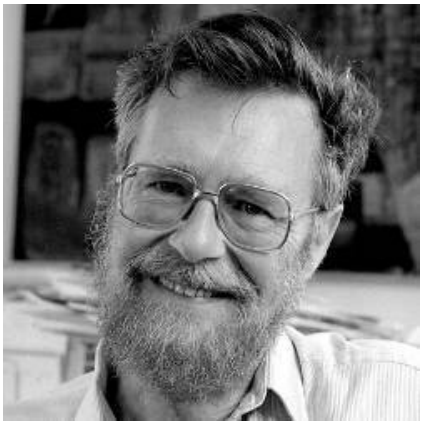


Figure 2.1: Edsger W. Dijkstra

2.1 Dijkstra's Shortest Path Algorithm

Algorithm 2.1.1 (Dijkstra's Shortest Path Algorithm)**.**

Set $\mathcal{S} = \emptyset$
Set $d(s) = 0$ and $d(v) = \infty, \quad \forall v \in V \setminus \{s\}$
While $\mathcal{S} \neq V$
  If $v = \underset{u \in V \setminus \mathcal{S}}{\operatorname{argmin}} \, d(u)$ then set $\mathcal{S} \leftarrow \mathcal{S} \cup \{v\}$
  For each arc (v, w):
    If $d(w) > d(v) + \ell_{vw}$ and $w \in V \setminus \mathcal{S}$
      then set $d(w) = d(v) + \ell_{vw}$

This algorithm only finds the shortest path distances. But, we can easily enhance the algorithm to keep track of the shortest path themselves as well:

Algorithm 2.1.2.

Set $\mathcal{S} = \emptyset$
Set $\mathcal{T} = \emptyset$, $\text{pred}(s) \leftarrow$ " $-$ ", and $\text{pred}(v) \leftarrow \square$
Set $d(s) = 0$ and $d(v) = \infty, \quad \forall v \in V \setminus \{s\}$
While $\mathcal{S} \neq V$
  If $v = \underset{u \in V \setminus \mathcal{S}}{\operatorname{argmin}} \, d(u)$
    then set $\mathcal{S} \leftarrow \mathcal{S} \cup \{v\}$ and $\mathcal{T} \leftarrow \mathcal{T} \cup (\text{pred}(v), v)$
  For each arc (v, w):
    If $d(w) > d(v) + \ell_{vw}$ and $w \in V \setminus \mathcal{S}$
      then set $d(w) = d(v) + \ell_{vw}$
      $\text{pred}(w) \leftarrow v$

See the example in the lecture and make sure you understand how this algorithm works!

2.1.1 Special Case reduced to BFS

If each arc-length is equal to 1, we simply get BFS. Try it!

2.1.2 Does it Work?

First observe that $\mathcal{S}$ is a set of vertices and $\mathcal{T}$ is a set of arcs. Thus, we introduce the following notation:

1. Let $\mathcal{S}^k$ denote the set of vertices in $\mathcal{S}$ at the end of the k-th iteration.
2. Let $\mathcal{T}^k$ denote the set of arcs in $\mathcal{T}$ at the end of the k-th iteration

Observe that the arcs in $\mathcal{T}^k$ are all between vertices in $\mathcal{S}^k$. Thus, $\mathcal{G}^k = (\mathcal{S}^k, \mathcal{T}^k)$ is actually a directed graph.

Remark. Note that $\mathcal{G} = \mathcal{G}^n$ is the final output of Dijkstra's algorithm.

In fact, it turns out that $\mathcal{G}$ is an **arborescence** (a directed tree) rooted at the source node s:

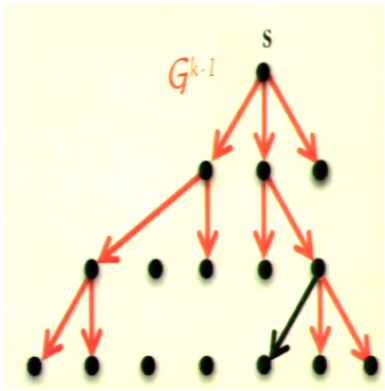
Theorem 2.1.1. *$\mathcal{G}^k$ is an arborescence rooted at s*

Proof. Base case is $k = 1$. Well, $\mathcal{S}^1$ only contains s. But $\mathcal{T}^1$ contains no arcs since $pred(s) =$ "$\_$". Thus, $\mathcal{G}^1$ is trivially an arborescence rooted at s.

Now, our hypothesis is that $\mathcal{G}^{k-1}$ is an arborescence rooted at s. Consider $\mathcal{G}^k$. Let v_k be the vertex added to $\mathcal{S}$ in the k-th iteration. Thus:

$$\mathcal{S}^k = \mathcal{S}^{k-1} \cup \{v_k\} \text{ and } \mathcal{T}^k = \mathcal{T}^{k-1} \cup (pred^{k-1}(v_k), v_k)$$

So, v_k has in degree =1 in $\mathcal{G}^k$ and out-degree = 0. Furthermore, by definition, $pred^{k-1}(v_k) \in \mathcal{S}^{k-1}$. Thus, $\mathcal{G}^k$ is an arborescnece rooted at s with v_k as a leaf.



□

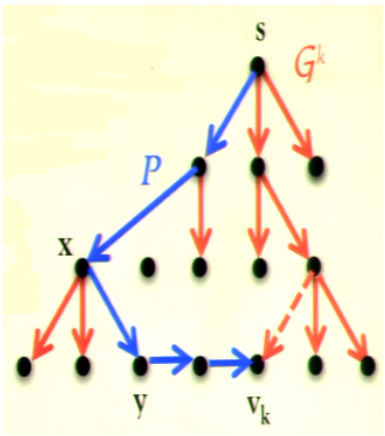
So, at this point, we know that Dijkstra's algorithm outputs an arborescence. But moreover, we claim that the paths induced by the arborescence are shortest paths:

Theorem 2.1.2. *The graph $\mathcal{G}^k$ gives the true shortest path distances from s to every vertex in $\mathcal{S}^k$*

Proof. The base case $k = 1$ is trivial since $d^1(s) = 0 = d^*(s)$. (Here we use the notation that d^* denotes the true shortest path distances.)

Now, for our hypothesis assume $\mathcal{G}^{k-1}$ gives the shortest distances from s to every vertex in $\mathcal{S}^{k-1}$ i.e. $d^{k-1}(v) = d^*(v)$ for all $v \in \mathcal{S}^{k-1}$.

Now, consider $\mathcal{G}^k$. Let v_k be the vertex added to $\mathcal{S}$ in the k-th iteration. We need to show that the new path is the shortest path to v_k. We argue via contradiction. Take the shortest path P from s to v_k with as many arcs in common with $\mathcal{G}^k$ as possible. Let x be the last vertex of $\mathcal{G}^{k-1}$ in P. Let $y \notin \mathcal{S}^k$ be the vertex after x in P. (Such a y $\exists$, or P is in $\mathcal{G}^k$ and we are done).



Since y is on the shortest path from s to v_k and the arc-lengths are non-negative, we have:

$$d^*(y) \leq d^*(v_k) < d^k(v_k)$$

noting that we have strict inequality or else we are done. But, since $x \in \mathcal{S}^{k-1}$ we have:

$$\begin{aligned} d^k(y) &\leq d^{k-1}(x) + l(x,y) \\ &= d^*(x) + l(x,y) \quad \text{by the induction hypothesis} \\ &= d^*(y) \end{aligned}$$

Therefore, $d^k(y) \leq d^*(y) < d^k(v_k)$ which contradicts the selection of v_k as $d^k(y) < d^k(v_k)$.

□

2.1.3 Run-time

We have n iterations and at most n distance updates at each iteration. Thus, the run-time is $O(n^2)$. Later in this course we will see how to implement Dijkstra's algorithm in $O(m \log n)$ using a heap (here m is the number of arcs).

3 Huffman Codes

Suppose we want to encode the alphabet in binary. How many bits do we need to be able to encode every letter? Well, 5 bits would suffice since $2^5 \geq 26$. But, is this good encoding? How do we measure the quality of this encoding? The most natural measure would be the length of the encoding. So if f_i is the frequency at which letter i appears in alphabet $\mathcal{A}$ then:

$$Cost = \sum_{i \in \mathcal{A}} l_i \cdot f_i$$

But, some letters appear more often than others in English, such as a and i. So, we could reduce the cost if more frequent letters have smaller representations than less frequent letters. Indeed, a well known code does exactly this:

3.1 Morse Code

By allowing a different number of bits per letter, only a max of 4 bits are required since $2^1+2^2+2^3+2^4 \geq 26$. However, we may think that there is a problem with Morse Code: its ambiguous. For example, $1101 = q$ but also $1101 = ma$. The way of fixing this is by using a "pause" or " $*$ " after each letter. This means that Morse Code is actually a ternary (not binary) code.

We are only interested in a binary code. So, is it possible to avoid ambiguities with a true binary code?

3.2 Prefix Codes

Definition 3.2.1. We say that a coding system is **prefix-free** if no code-word is a prefix of another code-word.

Remark. Morse code is NOT prefix free.

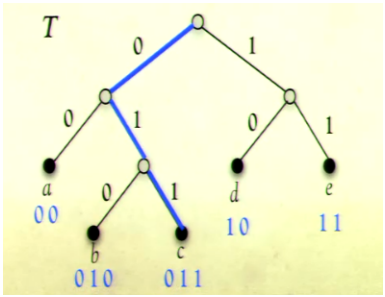
3.2.1 Binary Tree Representations

We can actually use a binary tree to represent a prefix-free binary code:

1. We have a binary tree T.
2. Each left edge has label 0 and each right edge has label 1.
3. The leaf vertices are the letter of the alphabet.

4. The code-word for a letter is the labels on the path from the root to the leaf.

Example 3.2.1.



Theorem 3.2.1. *A binary coding system is prefix-free iff it has a binary tree representation.*

Proof. ■ ←: The letters are the leaves, thus a path P_x from the root to leaf x and a path P_y to a leaf y must diverge at some point. Thus, the code-word for x cannot be a prefix of the code-word for y.

■ →: Given a binary coding system, we define the binary tree recursively. A letter whose code word starts with a 0 is placed in the left sub-tree, otherwise it is placed in the right sub-tree.

□

Our first important observation is the following:

Theorem 3.2.2 (Observation 1)**.**

$$Cost(T) = \sum_{i \in \mathcal{A}} f_i \cdot d_i(T)$$

where $d_i(T)$ denotes the depth of the leaf in the tree.

This first observation (theorem 3.2.2) tells us that if we must use a tree of specified shape then it is easy to determine the best way to assign the letter to the leaves. Specifically, sort the letter in increasing order of frequency then iteratively assign the least frequent letter to the deepest leaf.

What Observation 1 (theorem 3.2.2) doesn't tell us though, is what the best shape for the tree is. So, let

$$n_e = \sum_{i \in \mathcal{A}: e \in P_i} f_i$$

be the number of letters (weighted by frequency) whose root-leaf paths use edge e in T. Then, we have the following:

Theorem 3.2.3 (Observation 2)**.**

$$Cost(T) = \sum_{e \in T} n_e$$

Proof.

$$\begin{aligned} Cost(T) &= \sum_{i \in \mathcal{A}} f_i \cdot d_i(T) \\ &= \sum_{i \in \mathcal{A}} f_i \sum_{e: e \in P_i} 1 \\ &= \sum_{e \in T} \sum_{i \in \mathcal{A}: e \in P_i} f_i \\ &= \sum_{e \in T} n_e \end{aligned}$$

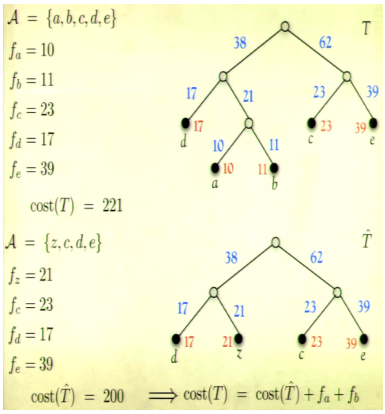
□

Also, we have the key formula:

Theorem 3.2.4 (Key Formula)**.** *Let $\hat{T}$ be the tree formed from T by removing a pair of sibling-leaves a and b and labelling their parent z where $f_z = f_a + f_b$. Then*

$$Cost(T) = Cost(\hat{T}) + f_a + f_b$$

Example 3.2.2.



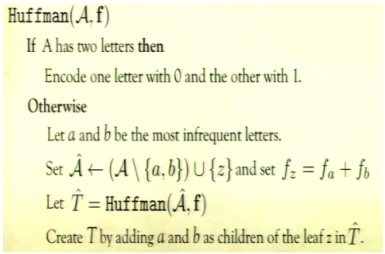
So now at this point we have:

1. Observation 1 $\rightarrow$ tells us that the two least frequent letters should be sibling leaves.
2. Observation 2 $\rightarrow$ tells us how to then recursively find the optimal shape of the tree.

We are now ready to introduce the main topic of this chapter:

3.3 Huffman's Algorithm

Algorithm 3.3.1 (Huffman's Algorithm)**.**



See the example in the lecture to understand how this algorithm works!!!

Theorem 3.3.1. *The Huffman Coding algorithm 3.3.1 gives the minimum cost encoding.*

Proof. The base case is $|\mathcal{A}| = 2$. Here both letters have code-words of length 1, so no improvement can be made.

Now, our hypothesis is to assume that Huffman's Algorithm works if $|\mathcal{A}| = k$. Assume that $|\mathcal{A}| = k + 1$ and let a, b be the least frequent letters. Then, a, b are siblings in an optimal solution, and for any $\hat{T}$, $Cost(T) = Cost(\hat{T}) + f_a + f_b$. So the best choice for $\hat{T}$ is the optimal solution for $\hat{\mathcal{A}}$. But by our hypothesis, this is exactly the choice made recursively by the algorithm.

□

3.3.1 Run-time

In this algorithm 3.3.1, there are $n-2$ iterations. In each iteration it takes $O(n)$ to find the two least frequent letters and update the alphabet. Thus, the run-time is $O(n^2)$. Later in the course we will use a heap to implement this algorithm in $O(n \log n)$.

4 The Minimum Spanning Tree Problem

Consider an undirected graph $G = (V, E)$ where each edge e has non-negative cost c_e. For convenience, assume all edge costs are distinct. Then the cost of a tree $T \subset E(G)$ is $Cost(T) = \sum_{e \in T} c_e$. The MST problem is to find a spanning tree T in G with minimum cost. An interesting thing about this problem is that most greedy algorithms work for this problem. We consider three examples of such algorithms.

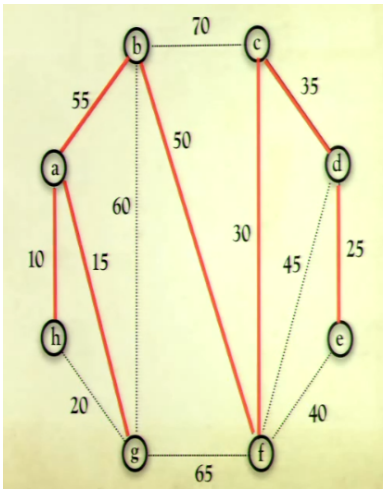
4.1 Kruskal's Algorithm

Algorithm 4.1.1 (Kruskal's Algorithm)**.** Sort the edges $\{e_1, ..., e_m\}$ by cost $c_1 < c_2 < ... < c_m$.
Set $T = \emptyset$
For $i = \{1, 2, ..., m\}$:

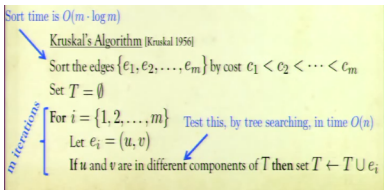
1. Let $e_i = (u, v)$
2. If u and v are in different components of T then set $T \leftarrow T \cup e_i$.

(i.e. we avoid cycles.)

Example 4.1.1.



4.1.1 Run-time



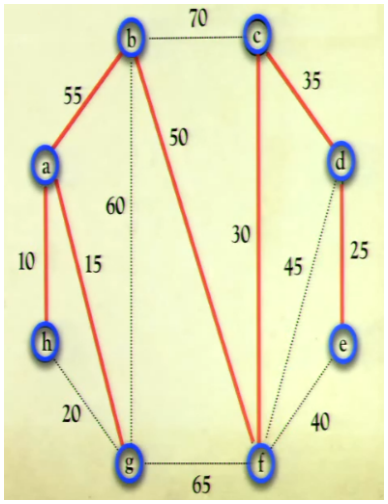
So, the run-time is $O(m \log m + mn) = O(mn)$.

4.2 Prim's Algorithm

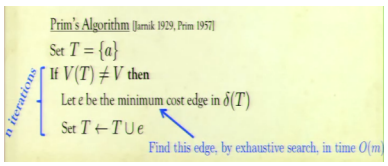
Algorithm 4.2.1 (Prim's Algorithm)**.** Set $T = \{a\}$.
If $V(T) \neq V$ then:

1. Let e be the minimum cost edge in $\delta(T)$
2. Set $T \leftarrow T \cup e$

Example 4.2.1.



4.2.1 Run-time



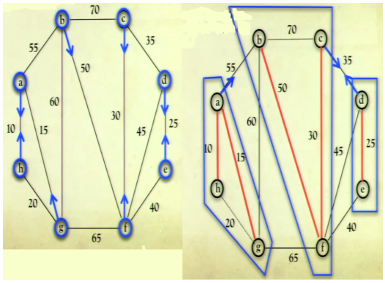
So, the run-time is $O(mn)$

4.3 Boruvka's Algorithm

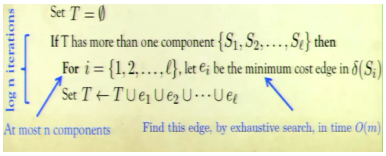
Algorithm 4.3.1 (Boruvka's Algorithm)**.** Set $T = \emptyset$
If T has more than one component $\{S_1, ..., S_l\}$ then:

1. For $i = \{1, 2, ..., l\}$ let e_i be the minimum cost edge in $\delta(S_i)$.
2. Set $T \leftarrow T \cup e_1 \cup e_2 \cup ... \cup e_l$

Example 4.3.1.



4.3.1 Run-time



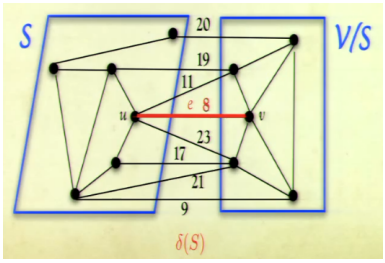
So, the run-time is $O(mn \log n)$

4.4 The Cut Property of MST's

We present and prove a property of MST's that can be used to prove that the above three algorithms all work.

Theorem 4.4.1 (Cut Property)**.** *Assume the edge costs are distinct. If e is the cheapest edge in some cut $\delta(S)$ then e is in the MST.*

Proof. We proceed via contradiction. Let $e = (u, v)$ be the cheapest edge in a cut $\delta(S)$.



Let T^* be a MST and assume $e \notin T^*$. Since T^* is a spanning tree, $\exists!$ path P in T^* from u to v.

Our first observation is that if $\hat{e} \in P$ then $(T^* \setminus \hat{e}) \cup e$ is a spanning tree.

Also, if you walk along P from u to v you must cross the cut $\delta(S)$ and so we have our second observation: $\exists$ at least one edge $\hat{e} \in P \cap \delta(S)$.

So now observation one and two yield $c_{\hat{e}} > c_e$. And thus $(T^* \setminus \hat{e}) \cup e$ is a cheaper spanning tree than T^*. Contradiction.

□

Now that we have proved the Cut Property (theorem 4.4.1), we trivially show that the above three algorithms work. Prim's and Boruvka's follow by definition since we have statements "let e be the minimum cost edge in $\delta(S)$". Now Kruskal's is a little more subtle. Recall that we had : If u and v are in different components of T then set $T \leftarrow T \cup e_i$. So let S be one of the two different components. Then e_i is the cheapest edge in $\delta(S)$ by the cost ordering.

4.5 Reverse-Delete Algorithm

We present one more algorithm for the MST problem that will connect to the next chapter.

Algorithm 4.5.1. Sort the edges $\{e_1, ..., e_m\}$ by cost $c_1 < c_2 < ... < c_m$.
For $i = \{m, m-1, ..., 2, 1\}$: If $G \setminus \{e_i\}$ is connected then set $G \leftarrow G \setminus \{e_i\}$

This algorithm is quite simple in practice, see the lecture for an example if needed.

4.5.1 Run-time

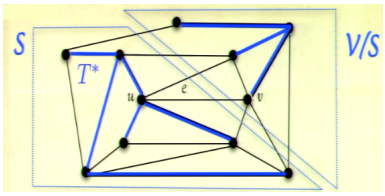
We have m iterations and can test connectivity in time $O(m)$ using some graph search algorithm (say DFS). Thus, the runtime is $O(m^2)$.

4.5.2 The Cycle Property of MST's

To prove that the Reverse-Delete algorithm works, we introduce another property of $MST's$:

Theorem 4.5.1 (Cycle Property)**.** *Assume distinct edge costs. If e is the most expensive edge in some cycle C then e is not in the MST.*

Proof. Let $e = (u, v)$ be the most expensive edge in the cycle C. So, $P = C \setminus e$ is a path from u to v.
Suppose e is in the MST T^*. Let $(S, V \setminus S)$ be the cut induced by $T^* \setminus e$.



Our first observation is that if $\hat{e} \in \delta(S)$ then $(T^* \setminus e) \cup \hat{e}$ is a spanning tree. Our second observation is that there is at least one edge $\hat{e} \in P \cup \delta(S)$.

So since e is the most expensive edge in cycle C we have $c_{\hat{e}} < c_e$. And thus, $(T^* \setminus e) \cup \hat{e}$ is a cheaper spanning tree than T^*. Contradiction.

□

Now we can prove that the Reverse-Delete algorithm works: We had "If $G \setminus \{e_i\}$ is connected then set $G \leftarrow G \setminus \{e_i\}$" but $G \setminus \{e_i\}$ means that e_i is in a cycle in G. So by the cycle property we are done.

5 Clustering

Given a collection of objects $\mathcal{O}$ we want to partition the objects into a set of clusters $\{S_1, ..., S_k\}$. Intuitively the basic objectives for a good clustering are:

1. Similar objects are placed in the same cluster

2. Dissimilar objects are placed in different clusters.

Anyways, we can actually represent the Clustering Problem by a weighted graph $G = (V, E)$:

1. There is a vertex for each object in $\mathcal{O}$
2. There is an edge between every pair of objects
3. The weight(length) $d_{ij} \geq 0$ of an edge $e = (i, j)$ is the dissimilarity between objects i, j

In this graph model we want vertices within a cluster to be close together and vertices in different clusters to be far apart. But how can we formally measure this quality of a clustering? Well unfortunately there is no optimal definition of this quality. We may be most interested in keeping similar points together. Or, we may be most interested in separating dissimilar points.

So, we restrict ourselves to a rather simple Clustering Problem:

5.1 Maximum Spacing Clustering

Here we want to maximize the distances between the clusters. We partition the vertices into k clusters so that the min. distance between two vertices in different clusters is maximized. Formally, given a clustering $\{S_1, ..., S_k\}$ we define the distance between two clusters as

$$d(S_l, S_m) = \min_{i \in S_l, j \in S_m} d_{ij}$$

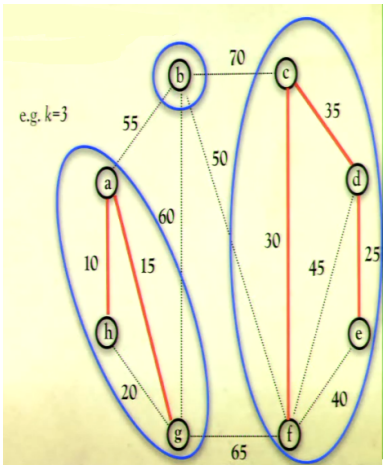
Then, our objective function for the MSC problem is

$$q(S) = \max_{S} \min_{l,m} d(S_l, S_m)$$

We can actually use the reverse delete algorithm to solve this problem.

Algorithm 5.1.1 (Reverse-Delete k-Clustering Algorithm)**.** Sort the edges by cost $c_1 < ... < c_m$
For $i = \{m, m-1, ..., 2, 1\}$: if $G \setminus \{e_i\}$ has k components or less then set $G \leftarrow G \setminus \{e_i\}$.
Output the k components $\{S_1, ..., S_k\}$ of G

Example 5.1.1.



5.1.1 Reverse-Delete Works

Here we prove that the Reverse-Delete k-Clustering Algorithm works.

Theorem 5.1.1. *A connected graph contains a spanning tree as a subgraph.*

Proof. A quick sketch: simply grow a BFS tree T from any root vertex. Since the graph is connected, tree T will contain every vertex.

□

Theorem 5.1.2 (Key Observation)**.** *When we remove an edge* $= (u, v)$*, the number of components increases by at most one.*

Proof. Originally, u, v are in the same component, say S_1. Since S_1 is a component, it contains a spanning tree T.

■ Case 1: $e \notin T$. Then S_1 remains a component after the deletion of e.

■ Case 2: $e \in T$ for every spanning tree in S_1. But, removal of an edge from a tree breaks it into exactly two subtrees. So, S_1 breaks into two components when we remove the edge e. And thus, the number of components increases exactly by one.

□

So finally,

Theorem 5.1.3. *The Reverse-Delete Clustering algorithm gives an optimal solution for the MSC problem.*

Proof. Let e_l be the edge whose deletion causes the number of components to increase from $k-1$ to k. Then this implies that the algorithm deletes all the edges $\{e_m, e_{m-1}, ..., e_{l+1}\}$ since there were at most $k-1$ components when they were examined.

So, when we delete e_l we obtain the clustering $\mathcal{S} = \{S_1, ..., S_k\}$. But, this means only edges in $\{e_m, e_{m-1}, ..., e_l\}$ can cross between two different clusters in $\mathcal{S}$. Thus e_l is the shortest edge between two different clusters in $\mathcal{S}$. Thus $q(\mathcal{S}) = d_{e_l}$. And so, our algorithm gives a k-clustering $\mathcal{S}$ with $q(\mathcal{S}) = d_{e_l}$. Furthermore the edges in $\{e_1, ..., e_l\}$ induce exactly $k-1$ connected components.

Any clustering $\mathcal{S}^* = \{S_1{}^*, ..., S_k{}^*\}$ with k components must separate the endpoints of at least one edge in $\{e_1, ..., e_l\}$. Thus $q(\mathcal{S}^*) \leq d_{e_l}$.

So, any other k-clustering S^* has $q(\mathcal{S}^*) \leq d_{e_l}$. So, we have the optimal solution.

□

6 The Set Cover Problem

We have a groundset $I = \{1, 2, ..., n\}$ of items and a collection $\mathcal{S} = \{S_1, ..., S_m\}$ of sets $S_i \subset I$ for each i. We want to select as few sets as possible such that every item is contained in at least one of the selected sets. In other words, we want to find the smallest collection of sets that covers the entire groundset.

Algorithm 6.0.1 (Greedy Set Cover Algorithm)**.** Repeat until $I = \emptyset$:

1. Let $\hat{S} = arg\max_{S \in \mathcal{S}} |S \cap I|$
2. Set $\mathcal{S} \leftarrow \mathcal{S} \setminus \hat{S}$ and $I \leftarrow I \setminus \hat{S}$

See the lecture recording to make sure you understand how this works in practice. We note that this greedy algorithm doesn't necessarily work since its not optimal. But, we will still use it since it is "OK".

6.1 Approximation Algorithms

Definition 6.1.1. An algorithm $\mathcal{A}$ is an α-approximation algorithm if for any instance I:

1. It runs in time $poly(|I|)$
2. It always outputs a feasible solution S
3. It always gives the approximation guarantee $Cost(S) \leq \alpha(Cost(Opt))$

So an approximation algorithm gives a quantitative performance guarantee. The Greedy Set Cover Algorithm is actually an approximation algorithm:

Lemma 6.1.1 (Observation 1)**.** If the optimal set cover has cardinality k, then for any $X \subset I$, there is some set in $\mathcal{S}$ that covers at least $\frac{1}{k}|X|$ items of X.

Proof. Let the optimal solution be $\{S_1{}^*, ..., S_k{}^*\}$. The sets $\{S_1{}^*, ..., S_k{}^*\}$ cover every item in I, thus they cover every item in $X \subset I$.

Since X is covered by k sets, there must be some set that covers a $\frac{1}{k}$ fraction of X.

□

Theorem 6.1.1. *If the optimal set cover has cardinality k, then our greedy algorithm finds a solution of cardinality at most $kln(n)$.*

Proof. WLOG let the greedy algorithm output $\{S_1, ..., S_T\}$. Let the optimal solution be $\{S_1{}^*, ..., S_k{}^*\}$. We want to show that $T \leq kln(n)$.

Let I_t be the uncovered items at the start of step t. Thus $I_1 = I$. Since $I_t \subset I$, by Observation 1, $\exists$ a set that covers at least $\frac{1}{k}|I_t|$ items in I_t $(\forall t)$. So, in step t, the greedy algorithm picks a set that covers at least $\frac{1}{k}|I_t|$ items in I_t. Therefore:

$$\begin{aligned}
|I_{t+1}| &\leq |I_t| - \frac{1}{k}|I_t| \\
&= \left(1 - \frac{1}{k}\right)|I_t| \ \forall t \\
&\leq \left(1 - \frac{1}{k}\right)\left(1 - \frac{1}{k}\right)|I_{t-1}| \\
&\dots \\
&\leq \left(1 - \frac{1}{k}\right)^t |I_1| \\
&= \left(1 - \frac{1}{k}\right)^t n \\
&< \left(e^{-\frac{1}{k}}\right)^t n \ \text{ since } \ 1 - x < e^{-x} \ \forall x \neq 0 \\
&= e^{-\frac{t}{k}} n
\end{aligned}$$

So setting $t = kln(n)$ gives $|I_{kln(n)+1}| < 1$. Therefore $|I_{kln(n)+1}| = 0$.

□

Corollary 6.1.1. The greedy algorithm is a $\log(n)$- approximation algorithm for the set cover problem.

This doesn't seem to be a good approximation guarantee. But actually,

Theorem 6.1.2. *There does not exist an approximation algorithm for the set cover problem with guarantee $(1 - o(1)) \log n$ unless $P = NP$. (See Comp 360)*

6.2 Run-time

We have n iterations. There are at most n coverage updates in each iteration. Thus, the run-time is $O(n^2)$.

7 The Greediest Algorithm and Matroids

We are given a set E of elements where each element has a weight $w_e \geq 0$. There is a collection $\mathcal{F}$ of feasible subsets of E and each set $F \in \mathcal{F}$ is a valid solution with weight $w(F) = \sum_{e \in F} w_e$. This induces the following general problem: "Find a feasible set in $\mathcal{F}$ with the maximum weight". This problem is too general, so we restrict ourselves to set systems with a natural property that arises in many problems such as interval selection and maximum weight spanning tree problems:

Definition 7.0.1. We say that $\mathcal{F}$ satisfies the hereditary property if:

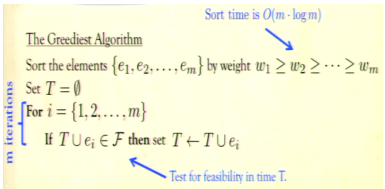
$$F \in \mathcal{F} \implies \hat{F} \in \mathcal{F} \; \forall \hat{F} \subset F$$

Then, we have the following generic algorithm for hereditary set systems:

7.1 The Greediest Algorithm

Algorithm 7.1.1 (The Greediest Algorithm)**.** Sort the elements $\{e_1, ..., e_m\}$ by weight $w_1 \geq w_2 \geq ... \geq w_m$.
Set $T = \emptyset$
For $i = \{1, 2, ..., m\}$: If $T \cup e_i \in \mathcal{F}$ then set $T \leftarrow T \cup e_i$.

7.1.1 Run-time



So the run-time is $O(m \log n + mT)$. So the algorithm is efficient if we can test for feasibility in polynomial time.

7.1.2 When Does the Greediest Algorithm Work?

Remark. Kruskal's algorithm 4.1.1 is a special case of the Greediest algorithm 7.1.1.

Remark. The Greediest algorithm 7.1.1 doesn't work for weighted interval selection.

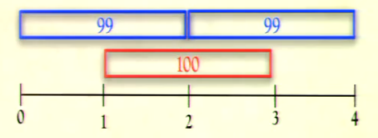


Figure 7.1: The greediest algorithm selects the red interval but clearly the optimal solution should choose both the blue intervals.

So when does the Greediest algorithm 7.1.1 work? What about the structure of the MST problem makes it compatible with the Greediest Algorithm?

Lets define one more property and then we can jump into the connection to matroids.

Definition 7.1.1. $\mathcal{F}$ satisfies the **augmentation property** if:

$$F, \hat{F} \in \mathcal{F} \ \& \ |F| > |\hat{F}| \implies \exists e \in F \setminus \hat{F} \ st \ \hat{F} \cup e \in \mathcal{F}$$

. In other words, given 2 feasible sets where one set is strictly larger than the other, then $\exists$ and element in the larger set that can be added to the smaller set whilst maintaining feasibility.

7.2 Matroids

Definition 7.2.1. A **matroid** is a nonempty set system $M = (E, \mathcal{F})$ that satisfies both the Hereditary Property and the Augmentation Property.

Remark. Hereditarity and non-emptiness implies $\emptyset \in \mathcal{F}$.

Matroids are actually very common.

Example 7.2.1. The graphical matroid: E is the set of edges in a graph. $F \in \mathcal{F}$ if F is a forest.

You should be able to, given a specific application, understand how the problem can be formulated in terms of a matroid problem!

Now, we lastly present a nice characterization of matroids which connects the concept to the Greediest Algorithm:

Theorem 7.2.1. *A hereditary nonempty set system M is a matroid iff the Greediest Algorithm outputs the optimal solution in M for any set of weights $\vec{w}$.*

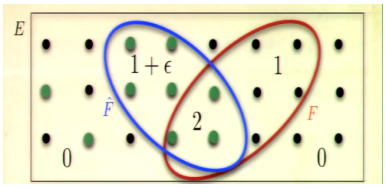
Proof. ■ $\rightarrow$: First, we prove that the greediest algorithm 7.1.1 always works on matroids.

Let the algorithm output $\{e_1, ..., e_l\}$ where $w(e_1) \geq w(e_2) \geq ... \geq w(e_l)$. Also, let the optimal solution be $\{{e_1}^*, ..., {e_k}^*\}$ where $w({e_1}^*) \geq ... \geq w({e_k}^*)$. Note that by the Augmentation Property, we have $l \geq k$, otherwise the algorithm could have selected one more element.

We claim that $w(e_i) \geq w({e_i}^*)$ for all $1 \leq i \leq l$. To prove this, we proceed via contradiction. Suppose not. Let j be the smallest index with $w(e_j) < w({e_j}^*)$. Consider the sets $\hat{F} = \{e_1, ..., e_{j-1}\}$ and $F = \{{e_1}^*, ..., {e_j}^*\}$. By the Augmentation Property, $\exists {e_i}^* \in F$ such that $\hat{F} \cup e_i \in \mathcal{F}$. But, $w({e_i}^*) \geq w({e_j}^*) > w(e_j)$. Thus we have a contradiction since the greediest algorithm 7.1.1 should have chosen ${e_i}^*$ in step j instead of e_j.

■ $\leftarrow$: Take a hereditary set system M that is not a matroid. So, $\exists F, \hat{F} \in \mathcal{F}$ with $|F| > |\hat{F}|$ but there doesn't exist $e \in F \setminus \hat{F}$ such that $\hat{F} \cup e \in \mathcal{F}$. We present a collection of weights that cause the greediest algorithm to fail:

$$w(e) = \begin{cases} 2 & e \in F \cap \hat{F} \\ 1 + \epsilon & e \in \hat{F} \setminus F \\ 1 & e \in F \setminus \hat{F} \\ 0 & e \notin F \cup \hat{F} \end{cases}$$



As the Greediest algorithm 7.1.1 runs,

1. It first selects all the elements in $F \cap \hat{F}$
2. It next selects all the elements in $\hat{F} \setminus F$
3. It finally (possibly) selects some elements in $E \setminus (F \cup \hat{F})$

So, the algorithm outputs a solution with weight

$$\begin{aligned} 2|F \cap \hat{F}| + (1+\epsilon)|\hat{F} \setminus F| + 0 &< 2|F \cap \hat{F}| + |F \setminus \hat{F}| \\ &= opt \end{aligned}$$

Therefore the greediest algorithm 7.1.1 does not work with these weights.

□

8 Acknowledgements

These notes have been transcribed from the lectures given by Prof. Adrian Vetta. Any figures are screenshots of the recorded lectures. This document is for personal use only and beware of typos!