CO 353 COURSE NOTES

COMPUTATIONAL DISCRETE OPTIMIZATION

KANSTANTSIN PASHKOVICH • WINTER 2023 • UNIVERSITY OF WATERLOO

Table of Contents

1 Shortest Paths

1.1 Preliminaries on Graphs

An **(undirected) graph** G is a pair (V, E), where E is a set of unordered pairs of elements in V. The elements of V are called **vertices** or **nodes**; the elements of E are called **edges**.

Let $u, v \in V$ and let $e = uv \in E$ be an edge.

- We say that e is **incident** to u and v.

- The vertices u and v are said to be **adjacent**.

- We call u and v the **endpoints** of e.

By default, we assume that there are no parallel edges (i.e. two edges $e = uv$ and $e' = u'v'$ in E with $\{u, v\} = \{u', v'\}$) and no loops (i.e. an edge $e = uv \in E$ with $u = v$).

For distinct $u, v \in V$, a u, v-**path** is a sequence of vertices $w_1, \dots, w_k$ such that $w_1 = u$, $w_k = v$, and $w_i w_{i+1} \in E$ for all $i = 1, \dots, k - 1$.

For example, consider the following graph $G = (V, E)$ with vertices $V = \{v_1, v_2, v_3, v_4\}$ and edges $E = \{v_1 v_2, v_1 v_4, v_2 v_3, v_2 v_4, v_3 v_4\}$.
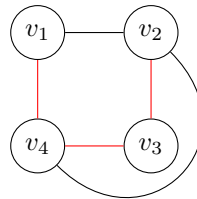
The lines in red form a v_1, v_2-path, namely v_1, v_4, v_3, v_2. Another v_1, v_2-path can be obtained by simply traversing the edge $v_1 v_2$.

A **cycle** in G is a sequence of vertices $w_1, \dots, w_{k+1}$ such that $w_i w_{i+1} \in E$ for all $i = 1, \dots, k$, the vertices $w_1, \dots, w_k$ are all distinct, and $w_1 = w_{k+1}$.

Finally, a graph G is **connected** if for any pair of distinct vertices $u, v \in V$, there exists a u, v-path in G.

1.2 Shortest Paths Problem

Given a *directed* graph $G = (V, E)$ with edge lengths $\ell_e \geq 0$ for each $e \in E$ and a distinguished start vertex $s \in V$, we wish to find shortest paths from s to every other vertex in V. Note that when we work with directed graphs, we will denote the directed edges with (v_1, v_2) as opposed to $v_1 v_2$ in the case of undirected graphs, where the order of the vertices did not matter.
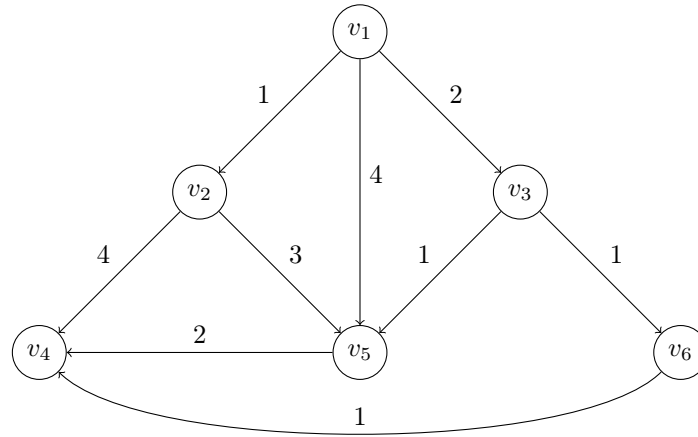
The **length** of a path P given by the sequence $w_1, \dots, w_k$ is given by

$$\ell(P) := \sum_{i=1}^{k-1} \ell_{(w_i, w_{i+1})} = \sum_{e \in P} \ell_e,$$

where the second sum makes sense because there are no parallel edges. Then the **shortest-path distance** from s to a vertex $u \in V$ is defined to be

$$d(u) := \min_{s,\, u\text{-paths } P} \ell(P).$$

For example, we can consider the following instance of an undirected graph with given edge lengths and starting vertex $s = v_1$.

In this case, we have $d(v_2) = 1$, since the only possible path from v_1 to v_2 is by taking the edge (v_1, v_2). There are multiple paths from v_1 to v_5; the shortest one is v_1, v_3, v_5 giving $d(v_5) = 3$.

Note that we always set $d(s) = 0$. We now make some observations:

(i) If $(u, v) \in E$, then $d(v) \leq d(u) + \ell_{(u,v)}$, since such an s, v-path is always an option.

(ii) For every $v \in V$ distinct from s, there exists $w \in V$ such that $d(v) = d(w) + \ell_{(w,v)}$ and $(w, v) \in E$. This can be seen by chopping off the last edge from a shortest path from s to v.

1.3 Dijkstra's Algorithm

In 1959, Dijkstra came up with the following algorithm to solve the shortest paths problem. The main idea is to maintain a set $A \subseteq V$ of "explored" nodes; that is, a set of nodes for which we already know the shortest-path distances. We'll also maintain labels $d'(v)$ for $v \in V \setminus A$ with upper bounds on the shortest-path distances from s.

Input. A directed graph $G = (V, E)$, edge lengths $\ell_e \geq 0$ for all $e \in E$, and a start vertex $v \in V$.

Output. For all $v \in V$, the length $d(v)$ for the shortest-path from s to v.

Step 1. **Initialization.** Set $A \leftarrow \{s\}$, $d(s) \leftarrow 0$, and $d'(v) \leftarrow \infty$ for all $v \in V \setminus A$.

Step 2. While $A \neq V$:

Step 2.1. **Push down the upper bounds.** For each $v \in V \setminus A$, compute

$$d'(v) \leftarrow \min \left\{ d'(v), \min_{\substack{u \in A \\ (u,v) \in E}} \{d(u) + \ell_{(u,v)}\} \right\}.$$

Step 2.2. **Add a new vertex.** Set $w \leftarrow \arg\min_{v \in V \setminus A} d'(v)$, $A \leftarrow A \cup \{w\}$, and $d(w) \leftarrow d'(w)$.

Suppose that for each vertex $w \in V$, we keep track of the node u determining its upper bound $d'(w)$. That is, the node u is such that $(u, w) \in E$ and $d'(w) = d(u) + \ell_{(u,w)}$. Then at the end of the algorithm, a shortest path from s to w can be obtained as a shortest path from s to u adjoined with the edge $(u, w) \in E$. Moreover, these edges selected by Dijkstra's algorithm form an arborescence, which is a nice graph structure that we'll discuss more later.

Next, let's prove the correctness of Dijkstra's algorithm. In particular, we need to show that for every $v \in V$, the distance from s to v is computed correctly. We'll assume that the graph is connected; that is, for every $v \in V$, there is an s, v-path in G. (Note that the algorithm won't terminate otherwise, but it can be adjusted to deal with this.)

Proof of correctness of Dijkstra's algorithm.

> We proceed by induction on $|A|$, and show that at each point in time, $d(v)$ is computed correctly for all $v \in A$. The case where $|A| = 1$ is clear because at the start of the algorithm, we initialize $A = \{s\}$ with $d(s) = 0$, which is correct.
>
> Assume that $d(v)$ is computed correctly for every $v \in A$ when that $|A| = k$. Suppose that we are adding a new vertex w to A in Step 2.2 of the algorithm. Consider the vertex $u \in A$ such that $(u, w) \in E$ and
>
> $$d'(w) = d(u) + \ell_{(u,w)}.$$
>
> Specifically, this is the vertex u determining the upper bound $d'(w)$ which we discussed in the paragraph following the description of the algorithm.
>
> For the sake of contradiction, assume that the distance from s to w is not $d'(w)$. Let P_u be a shortest path from s to u, and let P' be a shortest path from s to w. Then by our assumption, we know that
>
> $$\ell(P') < \ell(P_u) + \ell_{(u,w)} = d'(w).$$
>
> Now, let $x, y \in V$ be such that $(x, y) \in E$ lies on the shortest path P' from s to w, with $x \in A$ and $y \in V \setminus A$. (This exists because at some point, the path must exit A to get from s to w.) Then we obtain
>
> $$d'(y) \leq d(x) + \ell_{(x,y)} \leq \ell(P') < \ell(P_u) + \ell_{(u,w)} = d'(w),$$
>
> where the first inequality is because of how $d'(y)$ is computed in Step 2.1, and the second inequality is because the shortest path from x to y adjoined with the edge (x, y) is part of the path P', noting that $\ell_e \geq 0$ for all $e \in E$. But this contradicts our choice of $w = \arg\min_{v \in V \setminus A}\{d'(v)\}$ in Step 2.2 since $y \in V \setminus A$ but $d'(y) < d'(w)$. $\qquad\square$

The **running time** of an algorithm is the number of elementary operations that the algorithm performs as a function of the input size. The **input size** is the number of bits needed to specify the input.

- For example, in order to specify an integer $n \geq 0$ in binary, we require about $\log_2(n)$ bits.

- To specify a graph $G = (V, E)$ with integral edge lengths $\ell_e \geq 0$ for each $e \in E$, we require approximately $|V| + |E| + \sum_{e \in E} \log_2(\ell_e)$ bits. Note that we can specify an edge with two pointers to the endpoints.

We will need big-O notation to describe running time, because we are interested in the asymptotic behaviour of algorithms. For two functions $f : \mathbb{R}^+ \to \mathbb{R}^+$ and $g : \mathbb{R}^+ \to \mathbb{R}^+$, we say that $f(n) = O(g(n))$ if there exist constants $n_0 \in \mathbb{N}$ and $c \geq 0$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$. For example, we have $2n^2 + 1 = O(n^2)$ and $\log(n) = O(n)$. An algorithm is then considered **efficient** if its running time is bounded above by a polynomial function of the input size.

Let's consider the running time of Dijkstra's algorithm by looking at a naive implementation. For ease of notation, we will write $|V| = n$ and $|E| = m$. Note that in the case that $G = (V, E)$ is connected, we have $m \geq n - 1$.

- Step 1 can be performed using $O(n)$ operations since we are only assigning values for each vertex.

- In Step 2, there are at most n iterations.
  - Step 2.1 can be performed using $O(m)$ operations because each edge will participate in at most two comparisons throughout the entire iteration.
  - Step 2.2 takes $O(n)$ operations in order to determine the vertex of minimum upper bound.

Therefore, the running time of the naive implementation is $O(n + mn + n^2) = O(mn)$, which is polynomial in the input size.

We note that there are better implementations of Dijkstra's algorithm than the naive one that we have just stated. For Step 2.1, we can use m DECREASE-KEY calls and for Step 2.2, we can use n EXTRACT-MIN calls. In particular, by using Fibonacci heaps, DECREASE-KEY has running time $O(1)$ and EXTRACT-MIN has running time $O(\log n)$, which brings the total running time down to $O(m + n \log n)$.
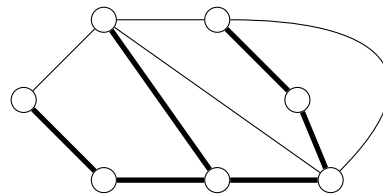
2   Minimum Spanning Trees

2.1   Trees

A **tree** is a connected acyclic graph; that is, a connected graph containing no cycles.
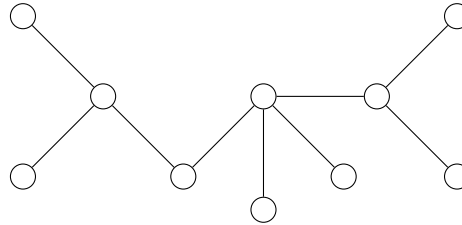
Given a graph $G = (V, E)$, a **spanning tree** of G is a graph $T = (V, F)$ such that $F \subseteq E$ and T is a tree. We illustrate an example of a graph G with a subtree T using bold edges below.

In an introductory graph theory course, such as MATH 239, it is shown that every tree on n vertices has $n - 1$ edges. The following theorem then gives us a useful characterization of trees.

THEOREM 2.1: FUNDAMENTAL THEOREM OF TREES

Let $T = (V, F)$ be a graph. The following are equivalent:

  (i) T is a tree.

  (ii) T is connected and $|F| = |V| - 1$.

  (iii) T is acyclic and $|F| = |V| - 1$.

In particular, if we know that two of the conditions hold, then the third one is guaranteed.

2.2   Minimum Spanning Trees

Given a connected graph $G = (V, E)$ and edge costs $c_e \geq 0$ for each $e \in E$, our goal is to find a spanning tree T of minimum cost

$$c(T) := \sum_{e \in T} c_e.$$

First, we'll set some notation. For a vertex $v \in V$, we define $\delta(v)$ to be the set of edges in E incident to v. More generally, given a subset of vertices $S \subseteq V$, the **cut induced by** S is defined to be the set

$$\delta(S) := \{uv \in E : u \in S, \, v \notin S\}.$$

The following theorem will be extremely important for finding a minimum spanning tree.

THEOREM 2.2: CUT PROPERTY

Suppose that the costs $c_e \geq 0$ for $e \in E$ are distinct. Let $S \subseteq V$ be such that $S \neq \varnothing$ and $S \neq V$, and let

$$e = \arg\min_{f \in \delta(S)} c_f.$$

Then every minimum spanning tree contains the edge e.

Proof of Theorem 2.2.

We proceed by contradiction. Let $S \subseteq V$ be such that $S \neq \varnothing$ and $S \neq V$, and let $e = \arg\min_{f \in S} c_f$. Suppose that there is a minimum spanning tree $T = (V, F)$ such that $e \notin F$.

Consider the graph $(V, F \cup \{e\})$. Note that $|F \cup \{e\}| = |V|$ and this graph is connected, so it cannot be a tree by Theorem 2.1. In particular, it must contain a cycle C.

Next, note that $|C \cap \delta(S)|$ must be even because for any edge leaving the cut $\delta(S)$, there must be another edge coming back into the cut. Moreover, since $e \in C \cap \delta(S)$, we have $C \cap \delta(S) \neq \varnothing$. This implies that $|C \cap \delta(S)| \geq 2$, so there is another edge $e' \neq e$ with $e' \in C \cap \delta(S)$.

Consider now the graph $T' = (V, (F \cup \{e\}) \setminus \{e'\})$. Then $|(F \cup \{e\}) \setminus \{e'\}| = |F| = |V| - 1$ and T' is connected because if a path between two vertices used the edge e', then we could go along the edges in $C \setminus \{e'\}$ instead. So by Theorem 2.1, T' is also a spanning tree. Finally, observe that

$$c(T) - c(T') = c_{e'} - c_e > 0$$

because we have $e = \arg\min_{f \in S} c_f$ and $e' \neq e$ with $e' \in \delta(S)$, as well as the assumption that the edge costs c_e for $e \in E$ were distinct. But T' has lower cost than T, contradicting our assumption that T was a minimum spanning tree. $\qquad\square$

The following property relating cycles to minimum spanning trees can also be proved similarly to Theorem 2.2. The main idea is to try to create shortcuts using the edges in the cycle.

THEOREM 2.3: CYCLE PROPERTY

Let $G = (V, E)$ be connected with distinct edge costs c_e for $e \in E$. Let C be a cycle in G and let

$$e = \arg\max_{f \in C} c_f.$$

Then no minimum spanning tree contains the edge e.

2.3   Prim's Algorithm

Prim's algorithm takes the idea of the cut property (Theorem 2.2) and uses it to compute a minimum spanning tree starting from an arbitrary vertex $s \in V$. At each iteration of the algorithm, we will keep track of a set of a partial tree construction T and vertices $A \subseteq V$ that are connected to s in T.

Input. A graph $G = (V, E)$ and edge costs $c_e \geq 0$ for all $e \in E$.

Output. A minimum spanning tree of G.

Step 1. **Initialization.** Let $s \in V$ be an arbitrary vertex, then set $A \leftarrow \{s\}$ and $T \leftarrow \varnothing$.
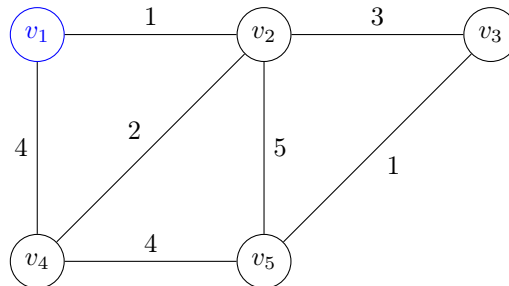
Step 2. While $A \neq V$:

Step 2.1. **Pick an edge for the minimum spanning tree.** Set

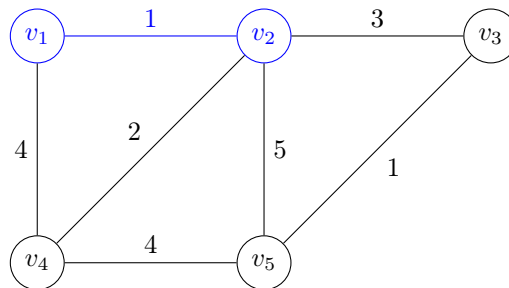$$e \leftarrow \arg \min_{f \in \delta(A)} c_f,$$

where $e = uv$ with $u \in A$ and $v \notin A$.

Step 2.2. **Update values.** Set $A \leftarrow A \cup \{v\}$ and $T \leftarrow T \cup \{e\}$.

Let's run Prim's algorithm on a simple example. Consider the following graph $G = (V, E)$, and suppose that at Step 1, we pick v_1 to be the initial vertex.

Then at the first iteration of Step 2.1, we have $\delta(A) = \{v_1 v_2, v_1 v_4\}$, so we pick $e = v_1 v_2$ because it has minimum cost. At Step 2.2, we set $A = \{v_1, v_2\}$ and $T = \{v_1 v_2\}$.

The edge $v_2 v_4$ is of minimum cost in the cut induced by A, so we set $A = \{v_1, v_2, v_4\}$ and $T = \{v_1 v_2, v_2 v_4\}$ at the next iteration. Continuing in this fashion, we obtain the following minimum spanning tree.
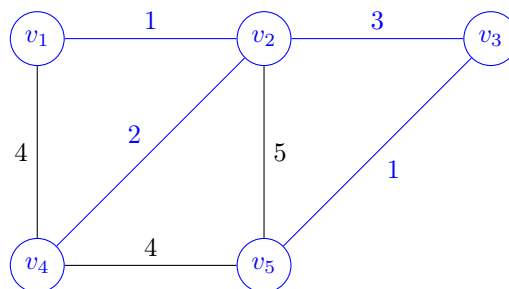
Note that Prim's algorithm may seen similar to Dijkstra's algorithm which also yields a spanning tree in the process of computing shortest paths. However, these algorithms fundamentally solve different problems, and there are examples where the resulting spanning trees do not coincide. Moreover, Dijkstra's algorithm takes a directed graph as input, whereas Prim's algorithm takes an undirected graph.

Next, let's prove that this indeed gives us a minimum spanning tree. For simplicity, we will assume that all edge costs c_e for $e \in E$ are distinct.

Proof of correctness of Prim's algorithm.

At every iteration of Step 2, we add one edge to T, and we run through $|V| - 1$ iterations. Therefore, T has exactly $|V| - 1$ edges. Moreover, an invariant of Prim's algorithm is that all vertices in A remain connected to s, so the final output is also connected. Therefore, we indeed obtain a spanning tree by Theorem 2.1. Finally, by the cut property (Theorem 2.2) and our assumption that all the edge costs are distinct, T contains only the edges that are in every minimum spanning tree, so T itself is a minimum spanning tree. □

As a consequence, we also have the following useful result.

COROLLARY 2.4

Let $G = (V, E)$ be a connected graph with distinct edge costs c_e for $e \in E$. Then G has a unique minimum spanning tree.

As usual, let's consider the running time, denoting $|V| = n$ and $|E| = m$ as before. For an efficient implementation, we keep track of a key $d'(v)$ for each $v \in V \setminus A$. Once a vertex w is added to A in Step 2.2, we update the keys via $d'(v) \leftarrow \min\{d'(v), c_{wv}\}$ for each $v \in V \setminus A$.

Note that Step 1 takes $O(1)$ time, and there are $n - 1$ iterations of Step 2. Implementing Prim's algorithm using priority queues, we note that each iteration of Step 2.1 involves one EXTRACT-MIN call, and going through all iterations of Step 2.2 takes at most m DECREASE-KEY calls in total. We recall that by using Fibonacci heaps, DECREASE-KEY is $O(1)$ and EXTRACT-MIN is $O(\log n)$, so we have a total running time of $O(n \log n + m)$.