Computational Discrete Optimization

CO 353

Chaitanya Swamy

LaTeXed by Sibelius Peng

Preface

Disclaimer Much of the information on this set of notes is transcribed directly/indirectly from the lectures of CO 353 during Winter 2022 as well as other related resources. I do not make any warranties about the completeness, reliability and accuracy of this set of notes. Use at your own risk.

Discrete optimization problems are underlying decisions that have a discrete flavor, e.g., YES/NO or $\{0,1\}$ decisions.

The focus in this course will be on algorithms, modelling. Broad classes of problems that we will study are network connectivity problems, location problems, general integer programs.

For any questions, send me an email via https://notes.sibeliusp.com/contact.

You can find my notes for other courses on https://notes.sibeliusp.com/.

Sibelius Peng

Contents

1

Graph Algorithms

1.1 Definitions, Notations & Terminology

A **graph** is a tuple (V, E), where V is set of **nodes/vertices**, E is set of **edges**, where edges **joins** two nodes.

If e is an edge that joins nodes u, v, then we denote this by $e = uv$. u, v are called **ends** of e. e is **incident** to nodes u, v. We are not allowing parallel edges, i.e., $e = uv$, and $e' = u'v'$ are distinct edges, then $\{u, v\} \neq \{u', v'\}$.

An *u*-*v* path in $G = (V, E)$ where $u, v \in V, u \neq v$, is a sequence of nodes $u_1 = u, u_2, \ldots, u_k, u_{k+1} = v$, where $u_i u_{i+1} \in E \ \forall i = 1, \ldots, k$. A **cycle** in G is a sequence of nodes $u_1, u_2, \ldots, u_k, u_{k+1} = u_1$ where $u_i u_{i+1} \in E \ \forall i = 1, \ldots, k$, and u_i's are distinct. Since there are no parallel edges, we can also identify a path/cycle by its sequence of $u_i u_{i+1}$ edges. So we will often refer to a path/cycle as a set of edges.

A graph G is **connected** if it has a $u - v$ path $\forall u, v \in V$ $(u \neq v)$. G is acyclic if G does not have a cycle. A **tree** is a connected, acyclic graph.

Let $G = (V, E)$ be a connected graph, and $T = (V_T, E_T)$ be a tree. IF $E_T \subseteq E$ and $V_T = V$, then we say that T is a **spanning tree** of G.

If C is a cycle, and $e \in C$, then $C - \{e\}$ still connects all nodes of C. So if G is a connected graph, and it contains a cycle C, and $e \in C$, then $G - \{e\} := (V, E - \{e\})$ is a connected graph. Hence, a spanning tree of G is a minimal connected subgraph of G. I.e., if $T = (V, F)$ where $F \subseteq E$ is a minimal set such that (V, F) is connected, then T is a spanning tree of G. If $T = (V, F)$ contains a cycle, then F is not minimal.

In **directed graph**, each edge has a direction, and goes **from** a node **to** another node.

1.2 Shortest paths: Dijkstra's algorithm

Problem Given a directed graph $G = (V, E)$ with edge costs $\{c_e \geq 0\}$ and a node $s \in V$, find the shortest path from s to all other nodes. The "shortest" path means path with the smallest total edge cost under the c_e edge costs.

Notation For a path P, let $c(P) := \sum_{e \in P} c_e$ denote the total cost of P. Let $d(u) = \min_{P: P \text{ a } s \to u \text{ path}} c(P)$, which is shortest path (SP) distance from s to u. If $u \to v$ is an edge of G, we have

$$d(v) \leq d(u) + c_{u,v} \qquad (\text{bat})$$

Dijkstra's Algorithm

The idea is to maintain a set of explored vertices, and we want to expand this set. Then we can make use of (bat) to estimate the shortest path from s to v, a vertex to be added to the set. We will maintain a label $\ell(v)$ for all $v \notin A$, which is our current estimate fo the $s \to v$ shortest path distance.

Given Directed graph $G = (V, E)$, $s \in V$, edge costs $\{c_e \geq 0\}$.

Algorithm 1: Dijkstra's Algorithm

Initialize $A \leftarrow \{s\}$, $d(s) = 0$, $\ell(v) \leftarrow \infty \; \forall v \notin A$.
while $A \neq V$ **do**
  For all $v \notin A$ such that $\exists u \in A$ with edge $u \to v$, update

$$\ell(v) = \min \left\{ \ell(v), \min_{u \in A:(u,v) \in E} \left(d(u) + c_{u,v} \right) \right\}$$

  Select $w \in V - A$ such that $\ell(w)$ has minimum $\ell(v)$ value among all $v \notin A$.
  Update $A \leftarrow A \cup \{w\}$, set $d(w) = \ell(w)$.

Remark:

Can obtain actual shortest paths by maintaining along with $\ell(w)$, the node $u \in A$ that determines $\ell(w)$ (i.e., $u \in A$ is s.t. $\ell(w) = d(u) + c_{u,w}$). Call u, the "parent" of w, and $u \to w$ the parent edge of w.

The shortest paths obtained via previous point have a special structure: every node $w \neq s$ has exactly one edge entering it, and there are no cycles, i.e., we have something like "directed" tree. And we denote shortest-path tree: directed tree returned by algorithm.

Also note that $\ell(v)$ in

$$\ell(v) = \min \left\{ \ell(v), \min_{u \in A:(u,v) \in E} \left(d(u) + c_{u,v} \right) \right\}$$

is redundant, since

$$\min_{u \in A:(u,v) \in E} \left(d(u) + c_{u,v} \right)$$

term only decreases as the set A only grows.

Correctness

We may assume that there exists $s \to u$ path in $G \; \forall u \in V$. And it's easy to modify Dijkstra's algorithm to detect if this assumption holds, and get shortest path distances from s to all nodes reachable from s.

Let $d^{\text{Alg}}(v)$: d-value computed by algorithm. Recall $d(v)$ is the shortest path distance from s to v. The goal then is to show that for all $v \in V$, $d^{\text{Alg}}(v) = d(v)$. Clearly this is satisfied when $v = s$.

Assume we have correctly computed shortest path distances for all $u \in A$, $\ell(v)$ is the length of the shortest path P such that *last edge of P (which enters v) comes from a node in A*.

Why? Consider such a path P. Let $u \to v$ be the last edge of P. So $u \in A$, $d^{\text{Alg}}(u) = d(u)$,

$$c(P) \geq d(u) + c_{u,v} = d^{\text{Alg}}(u) + c_{u,v} \geq \ell(v)$$

and last inequality is by the definition of $\ell(v)$.

Theorem 1.1

If w is added to A in line 5 of the algorithm, then $d^{\text{Alg}}(w) = d(w)$. (I.e., we have computed shortest path distance from s to w.)

Proof:
Assume we have correctly computed shortest path distance $\forall u \in A$. Consider an arbitrary $s \to w$ path P. Let u be the last node on P that lies in A. Let v be the node on P after u (so $r \notin A$). Let P' be the $s \to v$ portion of P. Then

$$c(P) \geq c(P') \geq d(u) + c_{u,v} = d^{\text{Alg}}(u) + c_{u,v} \geq \ell(v) \geq \ell(w)$$

where the last equality is by the definition of w in the line 4. □

Then following parent edges gives an $s \to w$ path of length = $\ell(w) = d^{\text{Alg}}(w)$.

1.3 Running time and Efficient Algorithms

The goal in this course is to design efficient algorithms. What does efficient mean? The short answer is "reasonable" running time.

Running time is number of elementary operations performed by algorithm as a function of input size. **Elementary operations** includes basic arithmetic (e.g., addition), comparisons (is $x < y$?), simple logical constructs (i.e., if-then-else), assignments. **Input size** is the number of *bits* needed to specify the input. Note that number of bits need to specify a number $x \geq 0$, x integer is roughly $\log_2 x$, which is much smaller than x it self.

For example, the size of an input of the Dijkstra's algorithm, $G = (V, E), \{c_e\}_{e \in E}$ is usually taken to be approximately $|V| + |E| + \sum_{e \in E} \log_2 c_e$.

Reasonable running time, i.e., efficient algorithm means that running time that is **polynomial function** of input size. In order to specify running time & input size in a convenient, compact way, we will use $O(\cdot)$ notation.

Given two functions: $f, g : \mathbb{R}_+ \mapsto \mathbb{R}_+$, we say that $f(n) = O(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

Here are some examples:

$$\begin{aligned} n &= O(n) \\ 2n + 10 &= O(n) \\ 3n &= O(n^2) \\ \alpha n^c + \beta &= O(n^d) \\ n \log_2 n &= O(n^2) \\ \log_2 n &= O(\log_{10} n) \\ 2^n &= O(3^n) \end{aligned}$$

$f(n) = O(1)$ means $f(n) \leq c$ for all $n \geq n_0$. $f(n) = O(n^{O(1)})$ is shorthand for $f(n)$ is bounded by some (fixed) polynomial function of n: $f(n) \leq d \cdot n^c$.

An algorithm with running time $f(n)$, where n is input size, is **efficient** if $f(n)$ is bounded by a polynomial function of n, i.e., $f(n) = O(n^{O(1)})$.

Now we can examine the running time of Dijkstra's algorithm (removing unnecessary $\ell(v)$ in line 3). Let $m = |E|$, $n = |V|$. We observe that there are n iterations of while loop. In each iteration:

1. Computing $\ell(v)$ takes $O(d^{\text{in}}(v))$ time where $d^{\text{in}}(v)$ is the number of edges entering v.
2. Computing $\ell(v)$ $\forall v$ takes $O(m)$ time since $\sum_{v \in V} d^{\text{in}}(v) = m$.
3. Line 4 takes $O(n)$ time.
4. Line 5 takes $O(1)$ time.

Each iteration takes $O(m+n)$ time. This is $O(m)$ if we assume there exists an $s \to v$ path $\forall v \in V$ since then $m \geq n-1$, so $n = O(m)$. Then the running time of algorithm is $O(mn)$ which is a polynomial function of input size.

However, we can have a better implementation. Observe that if $\{u \in A : (u, v \in E)\}$ does not change across iterations, then $\ell(v)$ does not change. So instead of recomputing $\ell(v)$ for all $v \notin A$, we do the following:

> When we pick $w \notin A$ to add to A, we only update $\ell(v)$ for all $v \notin A$ such that $(w, v) \in E$, and set $\ell^{\text{new}}(v) = \min(\ell^{\text{old}}(v), d(w) + c_{w,v})$.

So the steps inside of while loop change as: [Let w^* be the last node added to A. Initially $w^* = s$.]

(a) For every edge (w^*, v), where $v \notin A$, update

$$\ell(v) = \min(\ell(v), d(w^*) + c_{w^*,v})$$

and we call this `DecreaseKey` operation.

(b) Find $w \notin A$ with minimum $\ell(\cdot)$ value. We call this `ExtractMin` operation.

(c) Update $A \leftarrow A \cup \{w\}$, $d(w) = \ell(w)$, $w^* = w$.

Across all iterations, we examine each edge (u, v) at most once in step (a) above (in the iteration when $w^* = u, v \notin A$). So across all iterations, $\leq m$ `DecreaseKey` operations, $\leq n$ `ExtractMin` operations.

Then we can use a simple array to store $\ell(\cdot)$ values. Note that `DecreaseKey` is $O(1)$, and `ExtractMin` operation is $O(n)$. Thus the running time = $O(m + n^2) = O(n^2)$.

There exist data structures such as priority queue, under which `DecreaseKey` and `ExtractMin` take $O(\log n)$. Then the running time is then $O(m \log n)$.

There exists a data structure called Fibonacci heaps, under which `DecreaseKey` is $O(1)$, and `ExtractMin` operation is $O(\log n)$. Then the running time is $O(m + n \log n)$.