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

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

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

4 Select $w \in V - A$ such that $\ell(w)$ has minimum $\ell(v)$ value among all $v \notin A$.
5 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 $O(m + n \log n)$.

2

Graph Algorithm cont'd

2.1 Minimum Spanning Trees

MST Problem Given a connected, undirected graph $G = (V, E)$, edge costs $\{c_e\}_{e \in E}$. Find a spanning tree of G of minimum total edge cost.

We say "T is a spanning tree" is equivalent to "T is the edge set of a spanning tree". We denote the cost of T by $c(T) := \sum_{e \in T} c_e$.

> **Note:**
> The c_e's could be positive, zero, or negative.
>
> If all c_e's are ≥ 0, then can equivalently define the MST problem as: find the min-cost connected spanning subgraph of G. Because there is always an optimal solution that is minimal connected spanning subgraph of G.

Theorem 2.1: Fundamental Theorem about trees

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

(a) T is a tree (i.e., connected, acyclic)

(b) T is connected, has $n - 1$ edges.

(c) T is acyclic, has $n - 1$ edges.

Proof:

(a) $\Rightarrow$ (b) Pick some $r \in V$ as root node. Root T ar r, i.e., draw T as hanging off of r. For each $v \neq r$, there is a unique edge uv of T (incident to v) such that u is closer to r than v, and we call uv the parent edge of v.

These parent edges cover T, and number of parent edges $= n - 1$, since each $v \neq r$ has a unique parent edge.

(b) $\Rightarrow$ (a) T is connected. Let T' be a spanning tree of T. So by (a) $\Rightarrow$ (b), we know that T' has $n - 1$ edges. But T has $n - 1$ edges. So $T = T'$, so T is a tree. □

2.2 Cut property

Notation Let $v \in V$. $\delta(v)$ denotes the set of edges incident to v. Let $S \subseteq V$, $\delta(S) := \{uv \in E : u \in S, v \notin S\}$. In other words, $\delta(S)$ denotes the "boundary" of S.

Now we assume that all edges costs are distinct. Fix some node $s \in V$. Let $e \in \delta(S)$ have the smallest edge cost among edges in $\delta(S)$. Is e in some MST? In fact, e is in every MST.

cut

A cut is any partition $(A, V - A)$ of the vertex set V, where $A \neq \varnothing, A \subsetneq V$.

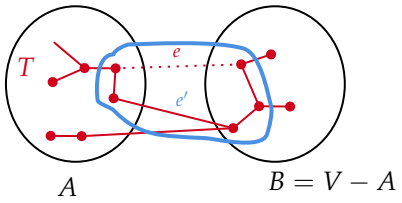
Edges crossing the cut are edges in $\delta(A)$ $(= \delta(V - A))$. If $F \subseteq E$, we say F crosses the cut to mean $F \cap \delta(A) \neq \varnothing$.

Lemma 2.2: Cut property

Consider any cut (A, B), where $B = V - A$. If e is the (unique) min-cost edge across the cut, then e belongs to every MST.

Proof (via an exchange argument):

Suppose T is an MST such that $e \notin T$. We will show that we can find another spanning tree T' (that contains e) such that $c(T') < c(T)$, then a contradiction.



$T \cup \{e\}$ contains a cycle C that contains e. And this is because $T \cup \{e\}$ is connected and has n edges, then it can't be acyclic. Here is a *basic fact*: if a cycle crosses a cut, it crosses the cut at least twice. So $\exists e' \in C \cap \delta(A)$, $e' \neq e$. By definition of e, $c_e < c'_e$. And $e' \in T$.

Consider $T' = T \cup \{e\} \setminus \{e'\}$. We claim that T' is a spanning tree. T' is connected since $e' \in$ cycle in $T \cup \{e\}$ and T' has $n - 1$ edges. Then we have

$$c(T') = c(T) + c_e - c'_e < c(T)$$

a contradiction. □

2.3 Prim's Algorithm

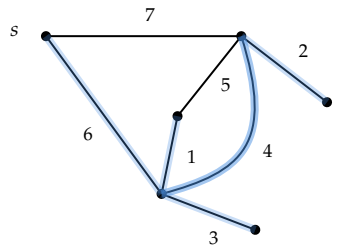
Now we can use cut property as the basis of the greedy algorithm.

Algorithm 2: Prim's Algorithm

```
1 Pick an arbitrary "seed" node s ∈ V.
2 Initialize A ← {s}, T ← ∅.
3 while A ≠ V do
4     Choose e = uv ∈ δ(A) with smallest cost, where u ∈ A, v ∉ A.
5     A ← A ∪ {v}, T ← T ∪ {e}.
6 return T
```

Example:

Blue lines are the output of Prim's algorithm.



Theorem 2.3

Prim's algorithm correctly computes an MST.

Proof:

Let T be the edge-set returned by Prim's algorithm.

- T is a spanning tree. T is connected, since every node is connected to s in T. Also, T has $n-1$ edges. Thus T is a spanning tree.
- T is a MST. Every $e \in T$ belongs to every MST by the cut property, since it is the min-cost edge across some cut. So $T \subseteq$ every MST. But T it self it a spanning tree, so T is MST.

□

Corollary

T is the unique MST.

If edge costs are not distinct, then Prim's algorithm still returns an MST; there could be multiple MSTs.

Implementation & Running Time

Implementation will be similar to Dijkstra's algorithm. For every unexplored node $v \notin A$, maintain a "key" $a(v) = \min_{e=uv:u\in A} c_e$. So in each iteration, we choose $w \notin A$ with smallest $a(\cdot)$ value similar to Dijkstra, let w^* be the last node added to A. In each iteration

(a) For each edge w^*v, where $v \notin A$, update $a(v) = \min\{a(v), c_{w^*v}\}$. `DecKey`

(b) Find $w \in V - A$ with smallest $a(\cdot)$ value. `ExtractMin`

(c) Set $A \leftarrow A \cup \{w\}$, and $T \leftarrow T \cup \{uw\}$, where $u \in A$, and $c_{uw} = a(w)$.

As in Dijkstra's algorithm, across all iterations:

1. n `ExtractMin` operations
2. m `DecKey` operations

So the running time is

- $O(m + n^2)$ using a simpel array to store keys (`DecKey` $O(1)$, `ExtractMin` $O(n)$)
- $O(m + n \log n)$ using a sophisticated data structure
  like Fibonacci Heaps (`DecKey` $O(1)$, `ExtractMin` $O(\log n)$)

2.4 Kruskal's algorithm

Kruskal's algorithm finds a MST. In some level, it is more greedy and intuitive than Prim's algorithm. The idea is to keep the edge costs in increasing order, and add edges to the set one by one, as long as no cycle are introduced.

Algorithm 3: Kruskal's algorithm

```
1 Sort the edges in increasing order of cost.
2 Initialize T ← ∅.
3 for each edge e in sorted order do
4     if T ∪ {e} does not have a cycle then
5         T ← T ∪ {e}
6 return T
```

Theorem 2.4

Kruskal's algorithm returns the unique MST when all edges costs are distinct.

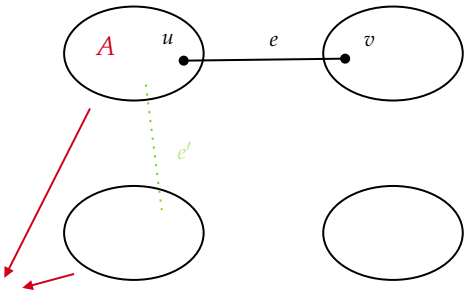
Proof:
Let T be the edge-set returned by Kruskal. Then, T is acyclic: by construction. Consider a basic fact: a graph is connected $H = (V_H, E_H)$ if and only if $\delta_H(A) \neq \varnothing \; \forall A : \varnothing \neq A \subsetneq V_H$.

Suppose (V, T) is not connected. Then from the basic fact, $\exists A, \varnothing \neq A \subsetneq V$ such that $\delta(A) \cap T = \varnothing$. But G is connected, so $\exists$ some edge $e \in \delta(A)$. Then $T \cup \{e\}$ is acyclic. So consider the point when Kruskal considers edge e. Let $F \subseteq T$ be set of edges Kruskal has added until then. Then $F \cup \{e\}$ is acyclic, so Kruskal should have added e. Then $e \in T$, a contradiction. So T is a spanning tree.

Consider any edge $e = uv \in T$. Let

$$A = \{w \in V : w \text{ is connected to } u \text{ in } T \text{ at the point when } e \text{ is considered by Kruskal}\}$$

Graph $H = (V, T)$ at the point when Kruskal considers e



Connected components of H

We claim that e is the min-cost edge in $\delta(A)$. Observe that e is the first edge of $\delta(A)$ considered by Kruskal. Hence e is the min-cost edge in $\delta(A)$. By claim, $e \in$ every MST. So $T \subseteq$ every MST. But T it self is a spanning tree. So T is MST. □

Remark:
We can stop Kruskal when $|T| = |V| - 1$.

Running time Sorting m edges takes $O(m \log m)$. To check if $e = uv$ can be added, we need to check if u, v are in different components of (V, T) are that point. There exist data structures (e.g., Union-Find) for maintaining connected components that allow one to do this $O(\log n)$ time. So total time for step 3 is $O(m \log n)$. Since $m \leq n^2$, the total time for the algorithm is $O(m \log m + m \log n) = O(m \log n)$.

2.5 Application to Clustering

Clustering Given a set of objects, and some notion of similarity/dissimilarity between these objects, divide the objects into groups (called clusters) so that

1. Objects in the same group are "similar" to each other.
2. Objects in different groups are "dissimilar" to each other.

Maximum-Spanning Clustering Given a set $V = \{p_1, \ldots, p_n\}$ of objects/points, and pairwise distances $d(p_i, p_j) = d(p_j, p_i) \geq 0 \ \forall i, j \in [n]$. The goal is to partition V into k clusters $C_1, \ldots, C_k$ ($C_i \cap C_j = \varnothing \ \forall i \neq j$, $\bigcup_{i=1}^{k} C_i = V$). So as to maximize the minimum inter-cluster spacing, which is equivalent to minimum distance between a pair of points in different clusters. I.e., maximize

$$\min_{\substack{i,j \in [k] \\ i \neq j}} \min_{\substack{p \in C_i \\ q \in C_j}} d(p, q)$$

Algorithm 4: Single-Linkage Clustering

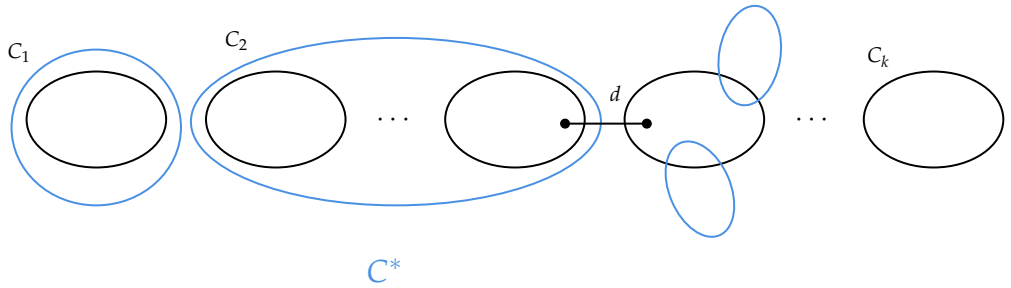
1 Start with every point in a separate cluster.
2 Repeatedly merge the 2 clusters with smallest inter-cluster distance[a], until we have k clusters.

[a] Distance between $C_i, C_j = \min_{p \in C_i, q \in C_j} d(p, q)$

This algorithm is called Single-Linkage Clustering, which is an example of an agglomerative algorithm (i.e., based on merging clusters). Consider a graph G with V as vertex set, and an edge between every pair $p, q \in V$, $p \neq q$, with cost $d(p, q)$.

> **Note:**
> Single-Linkage Clustering is exactly Kruskal (merging 2 clusters C_i, C_j due to points $p \in C_i, q \in C_j$) when adding edge pq. *Except* that we stop when there are k components. And this is the same as taking an MST and deleting the $k-1$ most costly edges, i.e., the $k-1$ edges that Kruskal could have added last.
>
> Thus equivalently, Run Kruskal (on complete graph with vertex set V, $d(p, q)$ edge costs) but stop when k components remain, which is equivalent to take MST and delete $k-1$ most costly edges.

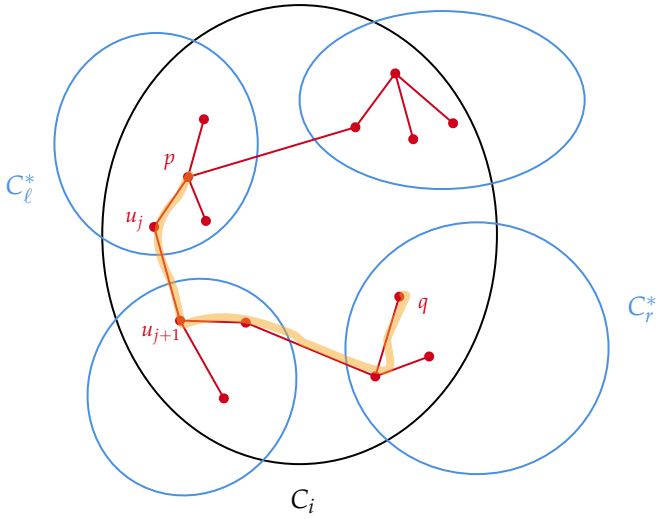
Now let's prove the correctness of Single-Linkage Clustering. Let $C_1, \ldots, C_k$ be clustering produced by MST - $\{k-1$ most costly edges$\}$. Let d be the spacing of this clustering.



Observe that

$$\begin{aligned} d &= \text{cost of edge Kruskal would have added next} \\ &= (k-1)\text{th most costly edge of MST} \end{aligned}$$

If $C = \{C_1, \ldots, C_k\}$ is not the optimum, let $C^* = \{C_1^*, \ldots, C_k^*\}$ be the optimum clustering. There might exist the case that $C_i \subseteq C_j^*$. As both C and C^* have k partitions, it's not possible to have $\subseteq$ for all k partitions. Since $C \neq C^*$, there is some cluster C_i that intersects at least two clusters of C^*.



Red edges inside C_i all have cost $\leq d$ since these are already added by Kruskal.

So there exists points p, q such that $p, q \in C_i$, but p, q lie in different clusters of C^*. Suppose $p \in C_\ell^*$, $q \in C_r^*$, $\ell \neq r$. Then considering $p - q$ path in C_i, there must be two consecutive nodes u_j, u_{j+1} such that $u_j \in C_\ell^*, u_{j+1} \notin C_\ell^*$. But then spacing of $C^* \leq d(u_j, u_{j+1}) \leq d$ since u_j, u_{j+1} are in different clusters of C^*. So this gives a contradiction since we assumed that $\{C_1, \ldots, C_k\}$ is not an optimal clustering, and optimal clustering has spacing strictly larger than d.

Index

C