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.

References:

[KT] Algorithm Design, Jon Kleinberg and Eva Tardos, Addison Wesley, 2005.

[V] Approximation Algorithms, Vijay Vazirani, Springer-Verlag, 2001.

[WS] The Design of Approximation Algorithm, David Williamson and David Shmoys, Cambridge University Press, 2011.

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

List of Algorithms

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

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

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 (🦇) 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 $v \notin A$). 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 Algorithms 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) ⇒ (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) ⇒ (a) T is connected. Let T' be a spanning tree of T. So by (a) ⇒ (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 $A \neq \varnothing, A \subsetneq V$.

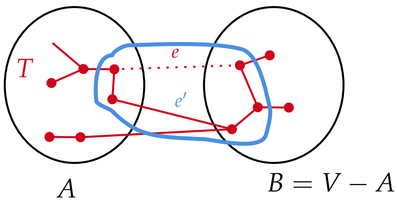
Edges crossing the cut are edges in $\delta(A)$ $(= \delta(V - A))$. If $F \subseteq E$, we say F crosses the cut to mean $F \cap \delta(A) \neq \varnothing$.

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. □

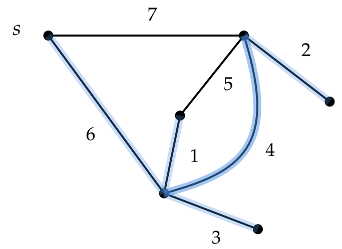
2.3 Prim's Algorithm

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

Algorithm 2: Prim's Algorithm

Pick an arbitrary "seed" node $s \in V$.
Initialize $A \leftarrow \{s\}$, $T \leftarrow \varnothing$.
while $A \neq V$ **do**
    Choose $e = uv \in \delta(A)$ with smallest cost, where $u \in A, v \notin A$.
    $A \leftarrow A \cup \{v\}$, $T \leftarrow T \cup \{e\}$.
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 $T \leftarrow \varnothing$.
3 **for** *each edge e in sorted order* **do**
4   **if** $T \cup \{e\}$ *does not have a cycle* **then**
5         $T \leftarrow T \cup \{e\}$
6 **return** T

Theorem 2.4

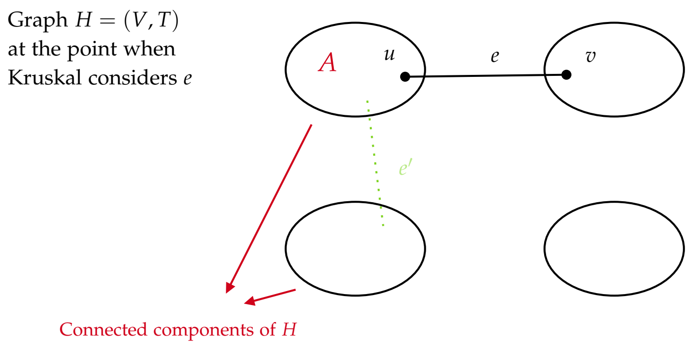
Kruskal's algorithm returns the unique MST when all edges costs are distinct.

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

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



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 $V = \{p_1, \dots, p_n\}$ 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 $C_1, \dots, C_k$ ($C_i \cap C_j = \varnothing \ \forall i \neq j$, $\bigcup_{i=1}^{k} C_i = V$). 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

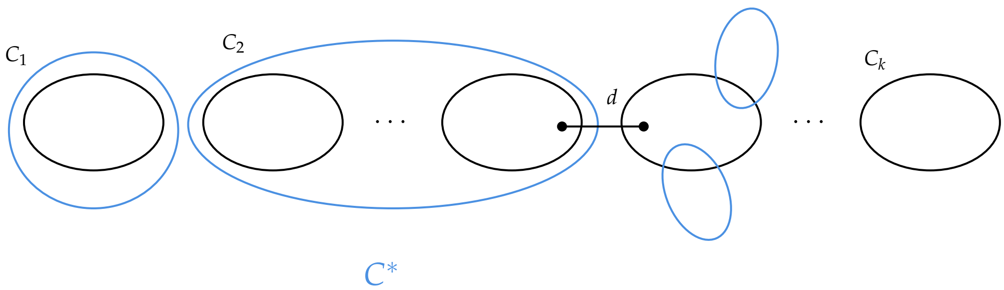
Start with every point in a separate cluster.
Repeatedly merge the 2 clusters with smallest inter-cluster distance[a], until we have k clusters.

[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.

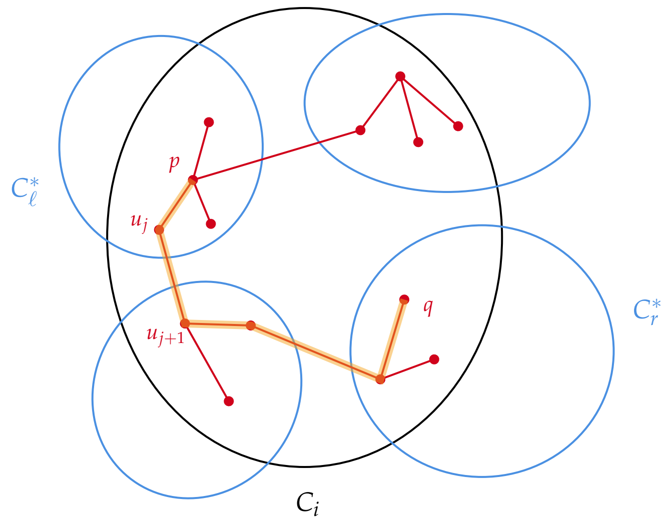
Now let's prove the correctness of Single-Linkage Clustering. Let $C_1, \dots, C_k$ be clustering produced by MST - $\{k-1$ most costly edges$\}$. Let d be the spacing of this clustering.



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 $C = \{C_1, \dots, C_k\}$ is not the optimum, let $C^* = \{C_1^*, \dots, C_k^*\}$ 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.

3

Graph Algorithms - cont'd

3.1 Arborescences

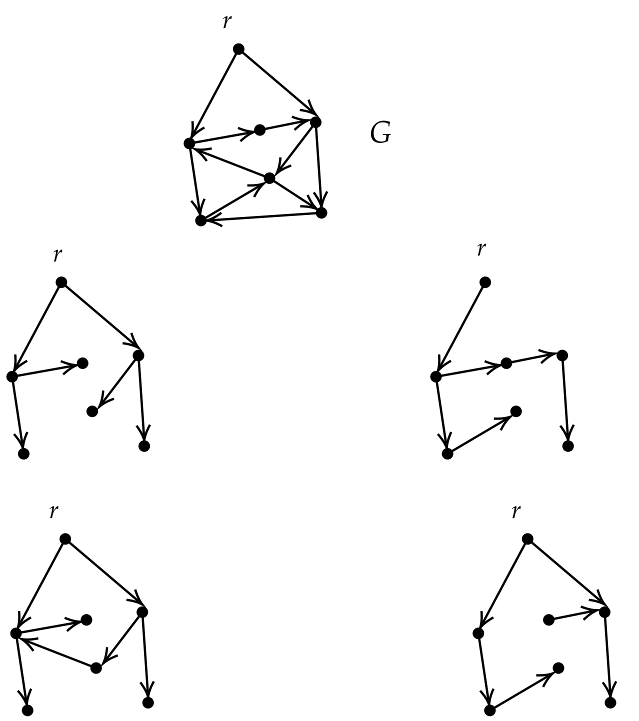
Arborescences is directed spanning tree. $G = (V, E)$ a directed graph, and let $r \in V$ be a "root" node.

arborescence rooted at r

An arborescence rooted at r (or rooted out of r) is a subgraph $T = (V, F)$ (so $F \subseteq E$) such that

- there is an $r \to v$ path in T $\forall v \in V$
- T is a spanning tree if we ignore the directions of the edges. I.e., "undirected version of T" is a spanning tree.

Example:



Arborscence

not an arborescence since undirected version contains a cycle

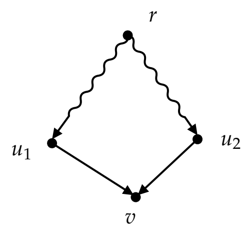
not an arborescence

Lemma 3.1: Useful alternate characterization of arborescence

$T = (V, F)$ is an arborescence rooted at r if and only if T has no (directed) cycles and every $v \neq r$ has exactly one incoming edge, (and r has no incoming edges).

Proof:

(⇒) Suppose T is arborescence. Since undirected version of T is a spanning tree, T has no directed cycles, and there exists $r \to v$ path in T for all $v \in V$ by the definition of arborescence. The path must be unique, otherwise we might encounter the situation like



then we will have a cycle in the undirected version of T. Hence every $v \neq r$ has exactly one incoming edge.

(⇐) Suppose T has no directed cycles, and every node $v \neq r$ has exactly one incoming edge. For any node $v \neq r$, we construct an $r \to v$ path as follows: we take v's unique incoming edge, say $u_1 \to v$, then u_1's unique incoming edge, and so on... Since T has no cycle, this process must stop because we have reached r, then we found an $r \to v$ path.

This also shows that undirected version of T is connected. Also, note that r has no incoming edges in T: suppose T had an edge $v \to r$, but then $r \to v$ path in T would create a directed cycle.

So we have shown that (since every $v \neq r$ has exactly one incoming edge) T has $n - 1$ edges. So undirected version of T is connected, has $n - 1$ edges, thus is a spanning tree. □

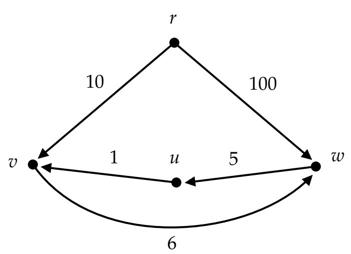
3.2 Min Cost Arborescence (MCA) Problem

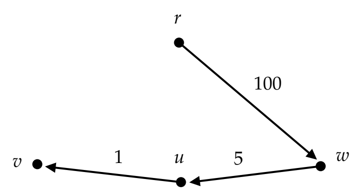
Given a directed graph $G = (V, E)$, and a root note $r \in V$, and edge costs $\{c_e\}_{e \in E}$, find an arborescence rooted at r of minimum total edge cost.

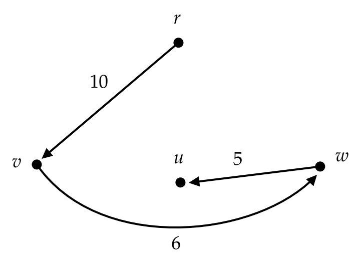
Assume there exists $r \to v$ path in G for all $v \in V$, then there exists an arborescence rooted at r.

There are two greedy strategies (inspired by MST)

Strategy 1 Pick the cheapest edge entering a node v (for some fixed node v). Does this edge belong to an MCA? Consider the following example for v. The cheapest edge entering v is of cost 1. However, if we pick this edge, the only arborescence containing edge $\vec{uv}$ is of cost 106, while MCA is of cost 21.







Strategy 2 Consider some cycle, and delete the most costly edge e of the cycle. Is this valid, i.e., is there an MCA not containing e? In the above example, we take the cycle $C = u \to v \to w \to u$. If we

delete the most cost edge vw, it's impossible to get MCA on the right.

Let us examine greedy strategy (1) again. Pick cheapest edge entering each node $v \neq r$. Let F^* be set of edges picked. Now let's make two observations.

Observation 1 If (V, F^*) is an arborescence, then it is an MCA. Every arborescence must pick an incoming edge for every $v \neq r$, and F^* is the cheapest way of picking these edges. If (V, F^*) is NOT an arborescence, then by Lemma 3.1, we know that (V, F^*) contains cycle Z, not containing root r.

Observation 2 Suppose for every node $v \neq r$, for each edge e entering v, we subtract a common amount P_v from its cost, i.e., $\forall v \neq r$, $\forall$ edges e entering v, we set $c'_e = c_e - P_v$. Then for any arborescence T, its c-cost and c'-cost differ by a constant; more precisely, $c(T) - c'(T) = \sum_{v \neq r} P_v$ (simply because T has exactly one edge entering each $v \neq r$ by lemma 3.1).

Corollary

Let $y_v = \min_{(u,v) \in E} c_{u,v}$. For all $v \neq r$, for all edges e entering v, let $c'_e = c_e - y_v$ [a]. Then T is an MCA with respect to $\{c_e\}$ if and only if T is an MCA with respect to $\{c'_e\}$ costs.

[a] $c'_e \geq 0$ for all e.

3.3 Edmond's algorithm

Now based on the corollary, imagine we have a cycle $Z \subseteq F^*$. And by observation 2, all edges of F^* (hence Z) have $c'_e = 0$. Then as long as we can reach one node of the cycle, we can take appropriate edges of the cycle to reach all nodes in the cycle without increasing the c'-cost. As we don't care about the inside of the cycle, we can contract the cycle and run the algorithm recursively.

Algorithm 5: Edmond's Algorithm for MCA

Input: Directed graph $G = (V, E)$, root $r \in V$, $\{c_e\}$ edge costs; assume there exists $r \to v$ path for all $v \in V$

if *G has only one node* **then return** $\varnothing$

foreach $v \neq r$ **do**

  $y_v := \min_{(u,v) \in E} c_{u,v}$

  $c'_e := c_e - y_v$ for all e entering v

For each $v \neq r$, choose a 0 c'-cost edge entering v. Let F^* be the resulting set of edges.

if *F^* is an arborescence* **then return** F^*

Otherwise F^* contains a directed cycle. Find a cycle $Z \subseteq F^*$ (not containing r). Contract Z into a single supernode [a] to get a graph $G' = (V', E')$.

Recursively find an MCA (V', F') in G' with respect to $\{c'_e\}$ edge costs.

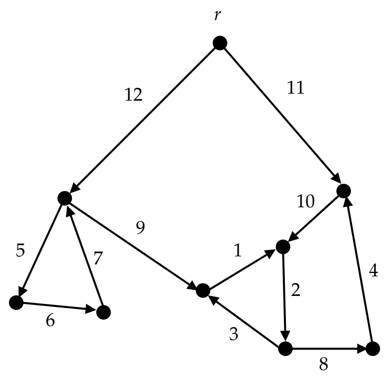
Extend (V', F') to an arborescence (V, F) in G:

- Let $v \in Z$ be the node that has an incoming edge in F'.
- Set $F \leftarrow F' \cup Z \setminus \{\text{edge of } Z \text{ entering } v\}$

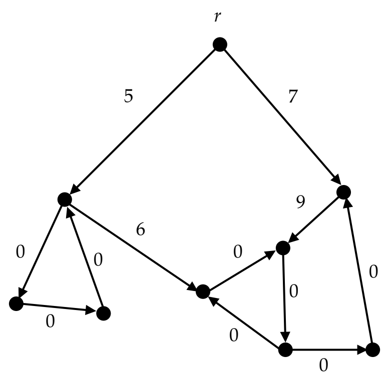
return F

[a] Remove self-loops, but retain parallel edges

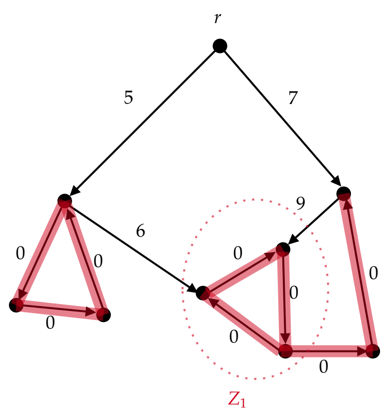
Now let's consider an example.



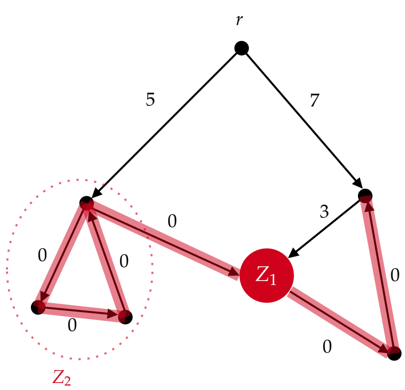
We then change the cost to c'_e.



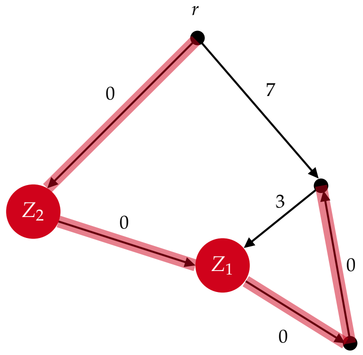
Then we mark the zero costs edges to produce F^*. And we can see there are cycles in this set of edges.



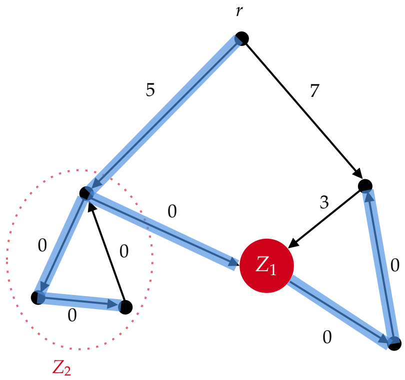
We first pick cycle Z_1 and contract it. And do the cost change again.



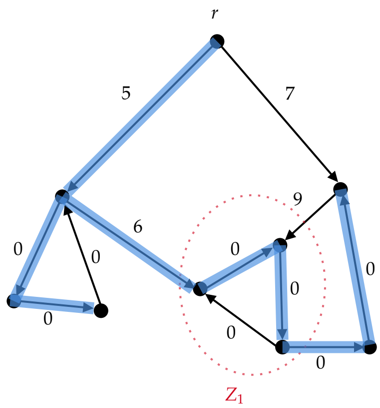
And we found cycle Z_2, and contract it, and do the cost change.



Now red edges are arborescence. Recursive calls stop. Then we expand it back.



Then expand again.



which is the output of the algorithm.

Running time

Line 0 takes $O(1)$. Line 1-2 are operations on edges, take $O(m)$. Line 3 on checking arborescence takes $O(n)$: follow incoming edges in F^* to see if we can reach r, either rv path or a cycle. Line 4 takes $O(|Z|) = O(n)$. Line 6 takes $O(|Z|) = O(n)$.

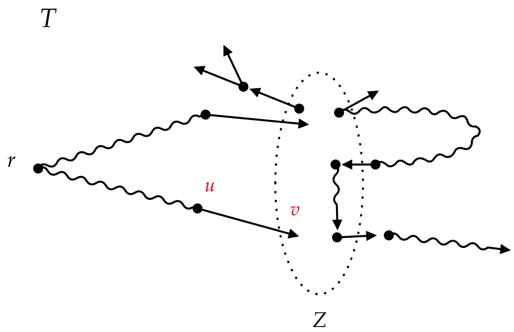
Algorithm makes $O(m)$ elementary operations and a recursive call to a smaller graph; at most n recursive calls. Thus polynomial running time.

Correctness

Lemma 3.2

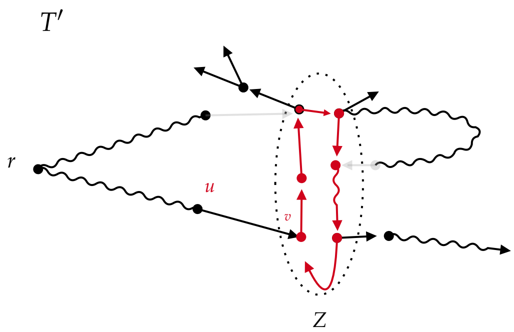
Suppose we have $\{d_e\}$ edge costs, and a 0 d-cost cycle Z such that $r \notin Z$. Then there exists an MCA (with respect to d-costs) that has exactly one edge entering Z.

Proof:
Let T be any arborescence, where it might has multiple edges entering Z. We will show that T can be modified to an arborescence T' with the stated property and such that $d(T') \leq d(T)$ where $d(T) := \sum_{e \in T} d_e$. T might look like the graph below.



Among all edges $(u', v') \in T$ that enter Z, let (u, v) be such that the $r \to u$ path in T has the fewest number of edges. Note that the $r \to u$ path in T has no nodes from Z. Let

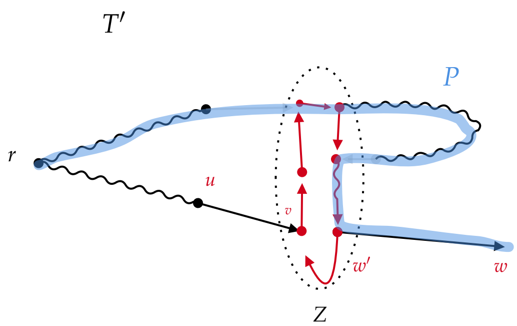
$$T' \leftarrow T - \{(u', v') \in T : v' \in Z, u' \notin Z, v' \neq v\} \cup \{\text{edges of } Z \text{ except for the edge entering } v\}$$



Note that $d(T') \leq d(T)$ because all edges we are adding in Z have zero costs. We now have to show T' is an arborescence.

T' has $n-1$ edges since every $w \neq r$ has exactly one incoming edge. Now it's remained to show that there exists an $r \to w$ path T' $\forall w \neq r$. Let P be the $r \to w$ path in T. If P does not contain any node of Z, then P is also an $r \to w$ path in T'.

Now suppose P contains a node of Z. Let w' be the last node of P in Z. Then T' contains:



- $r \to v$ path in T
- $v \to w'$ portion of Z
- $w' \to w$ portion of P

Concatenating these gives an $r \to w$ path in T'.

This also shows that undirected version of T' is connected, and has $n-1$ edges. Thus the undirected version is a spanning tree. □

Observation Any arborescence in G that enters cycle Z in step 4 of algorithm exactly once, yields an arborescence in G' and VICE VERSA, and these two arborescence have the same c'-cost.

Theorem 3.3

Edmond's algorithm finds an MCA of G.

Proof:
By induction on $|V|$. Base cases where $|V| = 1$ or 2 are clearly true. Suppose inductively, algorithm finds an MCA T' in G' with respect to c'-edge costs.

Let T^* be an MCA in G with respect to c'-edge costs that enters Z exactly once, which exists by lemma 3.2. Let $T^{*\prime}$ be arborescence obtained from T^* for graph G'.

We have

$$\begin{aligned} c(T) &= c'(T') && \text{since } c'_e = 0 \quad \forall e \in Z \\ &\leq c'(T^{*\prime}) && T' \text{ is an MCA of } G' \text{ wrt. } c'\text{-edge costs, induction hypothesis} \\ &= c'(T^*) && \text{edges of } Z \text{ have } 0\ c'\text{-cost} \end{aligned}$$

So T is an MCA of G with respect to c'-edge costs. Thus T is an MCA of G with respect to c-edge costs. □

4

Matroids

4.1 Introduction

matroid

A matroid is a tuple $M = (U, \mathcal{I})$, where U is a ground set (or universe), and $\mathcal{I} \subseteq 2^U$ is a collection of subsets of U satisfying the following properties:

(a) $\varnothing \in \mathcal{I}$.

(b) If $A \in \mathcal{I}$, and $B \subseteq A$, then $B \in \mathcal{I}$.

(c) (Exchange property) if $A, B \in \mathcal{I}$, with $|A| < |B|$, then $\exists e \in B - A$ such that $A \cup \{e\} \in \mathcal{I}$.

Sets in $\mathcal{I}$ are called **independent sets**. A set not in $\mathcal{I}$ is called a **dependent set**.

A **maximal independent set**, i.e., a set $B \in \mathcal{I}$ such that $B \cup \{e\} \notin \mathcal{I}\ \forall e \in U - B$, is called a **basis**.

Example: uniform matroid

Let U be any n-element set, e.g., $\{1, \ldots, n\}$. Let $\mathcal{I} := \{A \subseteq U : |A| \leq k\}$, where $k \geq 0$. Then $M = (U, \mathcal{I})$ is a matroid.

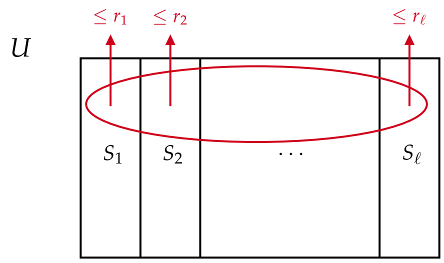
(a) $\varnothing \in \mathcal{I}$ since $|\varnothing| = 0 \leq k$

(b) If $A \in \mathcal{I}$ and $B \subseteq A$, then $|B| \leq |A| \leq k$, so $B \in \mathcal{I}$.

(c) Suppose $A, B \in \mathcal{I}$ with $|A| < |B| (\leq k)$. Take any $e \in B - A$, and note that $|A \cup \{e\}| = |A| + 1 \leq |B| \leq k$, so $A \cup \{e\} \in \mathcal{I}$.

Example: partition matroid

Again, let U be n-element set. Let $(S_1, \ldots, S_\ell)$ be a partition of U, and let $r_1, \ldots, r_\ell \geq 0$ be some non-negative integers. Let $\mathcal{I} := \{A \subseteq U : |A \cap S_i| \leq r_i \quad \forall i = 1, \ldots, \ell\}$.



Then $M = (U, \mathcal{I})$ is a matroid.

(a), (b) hold trivially. For part (c), suppose $A, B \in \mathcal{I}$ with $|A| < |B|$. Then there exists S_i such that $|A \cap S_i| < |B \cap S_i|$ (since $(S_1, \ldots, S_\ell)$ partition U). Consider any $e \in (B \cap S_i) - (A \cap S_i)$, and $A' = A \cup \{e\}$. Then

$$|A' \cap S_j| = \begin{cases} |A \cap S_j| \le r_j & \text{if } j \ne i \\ |A \cap S_i| + 1 \le |B \cap S_i| \le r_i & \text{if } j = i \end{cases}$$

So $A' \in \mathcal{I}$.

Example:

Let U be collection of n vectors in $\mathbb{R}^d$. Let

$$\mathcal{I} := \{A \subseteq U : \text{ the vectors in } A \text{ are linearly independent}\}$$

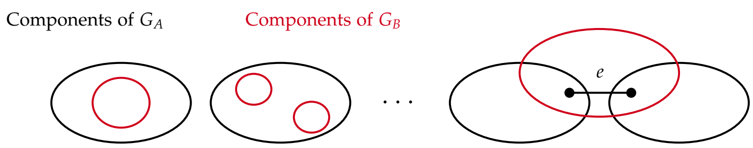
Then $M = (U, \mathcal{I})$ is a matroid.

(a), (b) hold trivially. (c) holds because of basic linear algebra: Suppose $A, B \in \mathcal{I}$ with $|A| < |B|$. If every $v \in B$ is a linear combinations of vectors in A, then since $|B| > |A|$, the vectors in B must be linearly dependent. Then there exists $v \in B$ such that $v \notin \text{span}(A)$ (where $\text{span}(A)$ is all vectors that are linear combinations of vectors in A). So $A \cup \{v\}$ consists of linearly independent vectors, i.e., $A \cup \{v\} \in \mathcal{I}$.

Example: graphic/cycle matroid

Let $G = (V, E)$ be an undirected graph. Let $U = E$, $\mathcal{I} := \{A \subseteq E : A \text{ is acyclic}\}$. Then $M = (U, \mathcal{I})$ is a matroid.

(a), (b) hold trivially. Consider (c). Suppose $A, B \in \mathcal{I}$ with $|A| < |B|$. Then $G_A = (V, A)$ has $n - |A|$ components where $n = |V|$, and $G_B = (V, B)$ has $n - |B| < n - |A|$ components.



In the picture above, we denote the components of G_B with red color. Some components of G_B can be contained within components of G_B. However, it cannot be that the vertex-set of every component of G_B is a subset of the vertex-set of some components of G_A. I.e., there exists some component of G_B that intersects at least two components of G_A as shown in the graph above.

This means there exists some $e \in B$ that connects two components of G_A. Thus $A \cup \{e\}$ is acyclic, so $A \cup \{e\} \in \mathcal{I}$.

4.2 Max-weight independent set (MWIS) problem

Here we present two matroid optimization problems.

Max-weight independent set (MWIS) problem

Given a matroid $M = (U, \mathcal{I})$, and weights $\{w_e\}_{e\in U}$ (the w_e's could be arbitrary), find a max-weight independent set, i.e., find $A \in \mathcal{I}$ such that

$$w(A) := \sum_{e\in A} w_e = \max_{B\in\mathcal{I}} w(B)$$

Matroid intersection problem

Given two matroids $M_1 = (U, \mathcal{I}_1)$ and $M_2 = (U, \mathcal{I}_2)$ and weights $\{w_e\}_{e\in U}$, find a max-weight set that is independent in both matroids (common independent set), i.e., solve

$$\max_{A\in\mathcal{I}_1\cap\mathcal{I}_2} w(A)$$

Matroid intersection problem is beyond the scope of this course, but you can learn it from CO 450.

Note that *max-weight spanning tree* with positive edge weights is a special case of MWIS problem, where the matroid is the graphic matroid. (since bases of graphic matroid associated with a connected graph are spanning trees). MST with $\{c_e\}$ edge costs can be captured by max-weight spanning tree with positive edge weights. By defining $w_e = M - c_e$, where $M > \max_e c_e$ (so that $w_e > 0$). For any spanning tree T $w(T) = (n-1)M - c(T)$.

Now we develop the greedy algorithm for MWIS. Input is $M = (U, \mathcal{I}), \{w_e\}_{e\in U}$. We may assume that $w_e > 0$ for all $e \in U$. Because otherwise, we can move to the smaller matroid

$$M' = (U' := \{e \in U : w_e > 0\}, \mathcal{I}' = \{A \subseteq U' : A \in \mathcal{I}\})$$

where we can verify M' is a matroid. Solving MWIS on M' will also solve MWIS on M. Since every independent set of M' is also independent in M. If $A \in I$, then $A \cap U' \in \mathcal{I}'$ and $w(A \cap U') \geq w(A)$.

Algorithm 6: Greedy Algorithm for MWIS

Input: $M = (U, \mathcal{I})$, $\{w_e\}_{e\in U}$
1. Sort elements in decreasing order of weight.
2. Initialize $A \leftarrow \varnothing$
3. Considering elements in sorted order, if $A \cup \{e\}$ is independent, then set $A \leftarrow A \cup \{e\}$ where e is current element being considered.
4. **return** A

Observe that above algorithm run on graphic matroid is equivalent to Kruskal for max-weight spanning tree.

Correctness

Claim 4.1

If B, B' are two bases of a matroid, then $|B| = |B'|$.

Proof:
Suppose not, and $|B| < |B'|$. Then since $B, B' \in \mathcal{I}$, by the exchange property of matroids, there exists $e \in B' - B$ such that $B \cup \{e\} \in \mathcal{I}$, contradicting that B is a maximal independent set. □

Now let A be the set returned by the greedy algorithm.

Claim 4.2

A is a basis of M.

Proof:
Suppose there exists $e \notin A$ such that $A \cup \{e\} \in \mathcal{I}$. Then consider the point when e is considered by greedy. At that point, we have some set $S \subseteq A$. But then $S \cup \{e\} \in \mathcal{I}$ since $S \cup \{e\} \subseteq A \cup \{e\}$, so algorithm should have added e, contradicting that $e \notin A$. □

Theorem 4.3

Greedy algorithm returns a max-weight independent set.

Proof:
Let A^* be a max-weight independent set. We want to show that $w(A) = w(A^*)$, hence A is a max-weight independent set.

Observe that A^* is a basis of M. Otherwise if there exists $e \notin A^*$ such that $A^* \cup \{e\} \in \mathcal{I}$, we have $w(A^* \cup \{e\}) > w(A^*)$. So we have $|A| = |A^*|$ by Claim 4.1 and Claim 4.2.

Let $k = |A| = |A^*|$. Suppose $w(A) < w(A^*)$. Let

$$A = \{e_1, e_2, \ldots, e_k\}$$
$$A^* = \{e_1^*, e_2^* \ldots, e_k^*\}$$

where elements are ordered by the ordering used by greedy.

Let $A_i := \{e_1, \ldots, e_i\}$ and $A_i^* := \{e_1^*, \ldots, e_i^*\}$ for all $i = 1, \ldots, k$. And we define $A_0 = A_0^* = \varnothing$. Consider the smallest index j such that $w(A_j) < w(A_j^*)$. Such j exists since $w(A_k) = w(A) < w(A^*) = w(A_k^*)$. We have

- $|A_j^*| = j$, $A_j^* \in \mathcal{I}$ since $A_j^* \subseteq A^*$
- $|A_{j-1}| = j - 1$, $A_{j-1} \in \mathcal{I}$ since $A_{j-1} \subseteq A$
- By the exchange property, there exists $e \in A_j^* - A_{j-1}$ such that $A_{j-1} \cup \{e\} \in \mathcal{I}$.

Since $e \in A_j^*$ and e_j^* has the least weight among elements of A_j^*, we have $w_e \geq w_{e_j^*}$. Since

$$w(A_{j-1}) + w_{e_j} = w(A_j) < w(A_j^*) = w(A_{j-1}^*) + w_{e_j^*}$$

and $w(A_{j-1}^*) \leq w(A_{j-1})$. Thus $w_{e_j^*} > w(e_j)$. To summarize, we have

$$w_e \geq w_{e_j^*} > w_{e_j}$$

Consider the point when greedy considers element e. At this point, we have some set $S \subseteq A_{j-1}$. This is because $w_e > w_{e_j}$, so greedy must consider e before e_j. But then $S \cup \{e\} \in \mathcal{I}$ since $S \cup \{e\} \subseteq A_{j-1} \cup \{e\}$ and so greedy should have added e. Contradiction, since then we would have $e \in A_{j-1}$. □

Running Time and Input Specification

How is the matroid M given as input? It would be space inefficient if we give all sets in $\mathcal{I}$ as input. So M is specified by means of the universe U and a **matroid independence oracle**, which is a procedure that given a set $S \subseteq U$ as input, answers (correctly) if $S \in \mathcal{I}$.

Running time is the number of elementary operations + number of calls (i.e., queries) made to independence oracle. Let $m = |U|$. Then the running time is $O(m \log m) + O(m \cdot \text{oracle}) = O(m \log m)$ which is polynomial-time, i.e., efficient algorithm.

4.3 Applications of Matroid Optimization

Maximum-Weight Bipartite Matching

bipartite

A graph $G = (V, E)$ is called bipartite if V can be partitioned as $L \cup R$ such that every edge has one end in L and one end in R. We call (L, R) bipartition of V.

matching

Given a graph $G = (V, E)$, a set $M \subseteq E$ is called matching if $|M \cap \delta(v)| \leq 1$ for all $v \in V$. I.e., for every $v \in V$, there is at most one edge of M incident to it.

Max-weight bipartite matching problem

Given a bipartite graph $G = (V, E)$, edge weights $\{w_e\}_{e \in E}$, find a matching M of maximum total edge weight.

Let $L \cup R$ be bipartition of V. Define the following two matroid, having ground set E:

$$M_L = \left(U = E, \mathcal{I}_L = \{A \subseteq U : |A \cap \delta(v)| \leq 1, \quad \forall v \in L\}\right)$$
$$M_R = \left(U = E, \mathcal{I}_R = \{A \subseteq U : |A \cap \delta(v)| \leq 1, \quad \forall v \in R\}\right)$$

Because the graph is bipartite, $\{\delta(v)\}_{v \in L}$ partitions E, $\{\delta(v)\}_{v \in R}$ partitions E. Then M_L and M_R are partition matroids. Thus $A \subseteq E$ is a matching if and only if $A \in \mathcal{I}_L \cap \mathcal{I}_R$. Hence max-weight matching is equivalent to max-weight independent set in M_L and M_R. I.e., max-weight bipartite matching can be solved using matroid intersection.

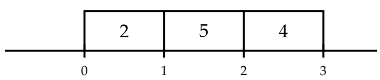
Schedling Problem

Given a set U of jobs; each $j \in U$ has **unit processing time**, deadline $d_j \geq 1$ which is integer, weights w_j (can be arbitrary). We can only process one job at any time. Find a max-weight set S of jobs, and an ordering of S such that all jobs in S complete by their deadlines.

For example,

j	1	2	3	4	5	6
d_j	3	2	1	3	2	4
w_j	2	1.8	5.6	-0.4	-1	3

Let $S = \{2, 5, 4\}$: can order S so that all jobs complete by their deadlines.



Consider $S = \{3, 2, 1, 4\}$. We cannot order S so that all jobs complete by their deadlines since there are four jobs in S with deadlines ≤ 3.

schedulable

Say that $S \subseteq U$ is "schedulable" if there exists an order of S that completes all jobs by their deadlines.

Exercise:
If S is schedulable, then ordering jobs in S in increasing order of deadlines yields an ordering where all jobs complete by their deadlines.

S is schedulable if and only if $\forall t = 0, 1, \ldots, |S|$, (number of jobs in S with deadline $\leq t$) $\leq t$

Theorem 4.4

$M = (U, \mathcal{I} = \{S \subseteq U : S \text{ is schedulable}\})$ is a matroid. Hence scheduling problem can be solved by solving MWIS for matroid M.

Proof:
Properties (a), (b) hold trivially. Consider exchange property. Let $A, B \in \mathcal{I}$ with $|A| < |B|$. We have a notation: for $S \subseteq U$, let $S_{\leq t} = \{j \in S : d_j \leq t\}$. Consider smallest $t \geq 0$ such that

$$|B_{\leq t'}| > |A_{\leq t'}| \qquad \forall t' \geq t \tag{$*$}$$

Such a t exists, since for $t = \max_{j \in B} d_j$, $(*)$ holds and for $t = 0$, $(*)$ does not hold, and $t \geq 1$.

Claim There exists $j \in B - A$ such that $d_j = t$.

If not, then all jobs in B with deadline equal to t are also in A. Then

$$\begin{aligned}
|B_{\leq t-1}| &= |B_{\leq t}| - (\#\text{ jobs in } B \text{ with deadline} = t) \\
&> |A_{\leq t}| - (\#\text{ jobs in } B \text{ with deadline} = t) && \text{due to } (*) \\
&\geq |A_{\leq t}| - (\#\text{ jobs in } A \text{ with deadline} = t) \\
&= |A_{\leq t-1}|
\end{aligned}$$

Contradicts t being the smallest value such that $(*)$ holds.

Claim $A' = A \cup \{j\}$ is schedulable where $d_j = t$.

Consider any $t' = 0, 1, \ldots, |A'|$. Then

$$|A'_{\leq t'}| = \begin{cases} |A_{\leq t'}| \leq t' & \text{if } t' < t \\ |A_{\leq t'}| + 1 \leq |B_{\leq t'}| \leq t' & \text{if } t' \geq t \end{cases}$$

So by exercise, A' is schedulable. So M is a matroid. □

5

Steiner Tree & Computational Complexity

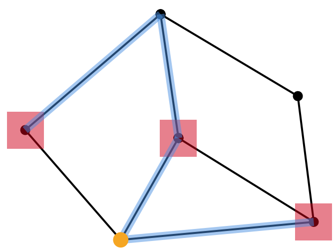
5.1 Minimum Steiner Tree Problem

Minimum Steiner Tree Problem

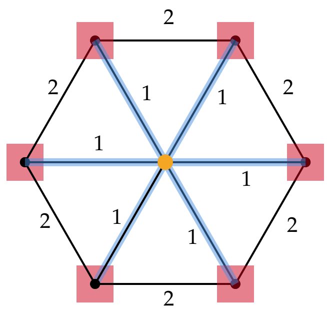
Given an undirected graph $G = (V, E)$, edge costs $c_e \geq 0$, and a set $T \subseteq V$ called **terminals**, find a min-cost tree that spans (i.e., connects) all the vertices in T.

This is called Steiner tree for T and we will omit "for T" if T is clear from the context.

Consider two examples.



Red boxes denote terminals T. Blue edges form a Steiner tree for T. Orange vertex is in Steiner tree but not in T. This is called a Steiner node. And we notice that this node is not necessary to include, but it might be useful in terms of the cost. For example,



Best Steiner tree containing only nodes from T has cost $2(|T| - 1)$. But blue edges form a Steiner tree of cost $|T|$.

Useful Transformation Let $G = (V, E)$. $(G, c, T) \to (G', c', T)$ where $G' = (V, E')$ is the complete graph on V and c'_{uv} is the shortest-path distance in G between u and v.

Note that $\forall u, v, w \in V$, $c'_{uw} \le c'_{uv} + c'_{vw}$ ($\triangle$-ineq) since one u-w path in G is shortest u-v path in G + shortest v-w path in G.

Claim 5.1

(a) Any Steiner tree F in G is also a Steiner tree in G', and $c'(F) \le c(F)$.

(b) Any Steiner tree F' in G' yields a Steiner tree F'' in G such that $c(F'') \le c'(F')$.

Proof:
For part (a), clearly F is a Steiner tree in G'. Also, note that $\forall uv \in E$, $c'_{uv} \le c_{uv}$, and so $c'(F) \le c(F)$.

For part (b), take F', and "expand" each edge $uv \in F'$ to the shortest u-v path P_{uv} in G, to get an edge-set $\hat{F}$. (So $\hat{F} = \bigcup_{uv \in F'} P_{uv}$, where P_{uv} is the shortest u-v path in G.)

Clearly $\hat{F}$ connects all vertices in G; but it could contain cycles. We get F'' from $\hat{F}$ by simply removing edges from $\hat{F}$ so that we get an acyclic graph. So

$$c(F'') \underset{\substack{\uparrow \\ \text{Since we removed} \\ \text{edges from } \hat{F} \text{ to set } F''}}{\le} c(\hat{F}) \le \sum_{uv \in F'} c(P_{uv}) = \sum_{uv \in F'} c'_{uv} = c'(F')$$

□

(G', c') defined as above is called **metric completion** of (G, c).

By Claim 5.1, we can always solve Steiner tree problem on the metric completion of (G, c). The optimum values do not change, and optimum solution in the metric completion gives us back an optimal solution in the original instance and vice versa.

5.2 MST-based algorithm

Idea Find an MST in $G'[T] := (T, E'[T])$ [1] with respect to c'-costs and map this to a Steiner tree in G using Claim 5.1 (b).

Algorithm 7: Algorithm for Minimum Steiner Tree Problems

1 Given instance (G, c, T), consider the instance (G', c', T), where (G', c') is the metric completion of (G, c).
2 Find an MST in $G'[T] := (T, E'[T])$ with respect to c'-costs and map this back to a Steiner tree in G using Claim 5.1 (b).

This is indeed a polytime algorithm, because all operations, including mapping back and so on, are all polynomial.

Cost of our solution $\le$ MST(G', c', T), which c'-cost of MST in $G'[T]$. Let

$$\text{OPT} := \text{OPT}(G', c', T) = \text{OPT}(G, c, T)$$

where OPT(G', c', T) is c'-cost of an optimal solution for (G', c', T).

Theorem 5.2

(Cost of solution returned $\le$) MST$(G', c', T) \le 2 \cdot \text{OPT}$.

[1]subgraph of G' induced by T. It is also complete because G is complete

Proof:

Let F^* be optimal Steiner tree for (G', c', T). Pick some $r \in T$, root F^* ar r, and do a DFS traversal of F^* starting at r.

DFS traversal yields a tour Z (i.e., a closed walk or cycle with repeated notes) that visits all vertices of F^*, and has c'-cost $\leq 2c'(F^*) = 2 \cdot \text{OPT}$.

Will "shortcut" Z to get a cycle $\hat{Z}$ that visits every terminal exactly once. Suppose

$$Z : u_0 = r, u_1, u_2, \ldots, u_{k-1}, u_k = u_0 = r$$
$$\hat{Z} : u_0 = r, u_{i_1}, u_{i_2}, \ldots, u_{i_{\ell-1}}, u_{i_\ell} = u_0 = r$$

where for every j, u_{i_j} is the first terminal after $u_{i_{j-1}}$ in the tour Z that has not been visited before.

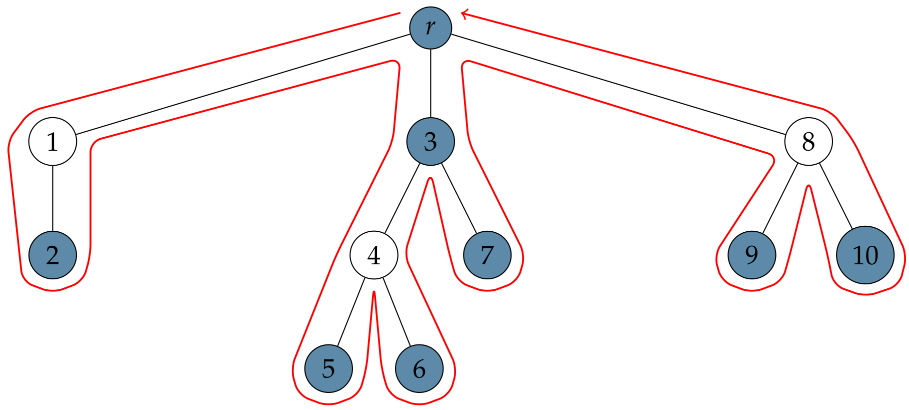
By $\triangle$-ineq, or the definition of c', we have $c'_{u_{i_j} u_{i_{j+1}}} \leq c'(Z_{u_{i_j} u_{i_{j+1}}})$ where $Z_{u_{i_j} u_{i_{j+1}}}$ is the portion of Z between first occurrences of u_{i_j} and $u_{i_{j+1}}$.

Since $Z_{u_{i_j} u_{i_{j+1}}}$ are disjoint for different j we have $c'(\hat{Z}) \leq c'(Z)$. Removing an edge from $\hat{Z}$ gives a spanning tree in $G'[T]$, of cost $\leq c'(\hat{Z}) \leq c'(Z) = 2 \cdot \text{OPT}$.

Thus $\text{MST}(G', c', T) \leq 2 \cdot \text{OPT}$. □

Example:

Here is an example of how we get $\hat{Z}$.

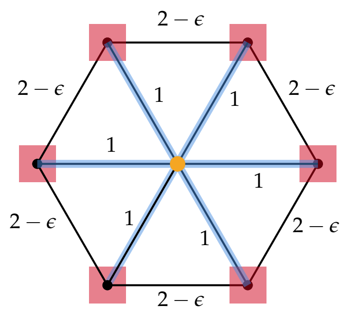


Terminal nodes are denoted by blue. So DFS traversal gives us Z

$$r, 1, 2, 1, r, 3, 4, 5, 4, 6, 4, 3, 7, 3, r, 8, 9, 8, 10, 8, r$$

We mark first occurrences of terminals in Z. And these marked vertices form $\hat{Z}$.

Were we dumb in analyzing this algorithm? Actually no. Consider an example.



where $\epsilon > 0$ is small. Our algorithm returns all edges of cost 2ϵ, thus the total cost is $(2 - \epsilon)(|T| - 1)$.

But the optimal solution is blue edges, of cost $|T|$. We see the factor is approximately 2. Can we design a better algorithm? For this problem, there is no known efficient algorithm that is always guaranteed to return an optimal solution. Moreover, researchers don't believe such an algorithm exists.

α-approximation algorithm

An α-approximation algorithm, where $\alpha \geq 1$ is the approximation factor, for a minimization problem, is an efficient (i.e., polytime) algorithm that on every instance returns a solution of cost $\leq \alpha \cdot \text{OPT}$.

So above algorithm for Steiner tree is a 2-approximation algorithm for the Steiner tree problem.

5.3 Computational Complexity

P is class/set of all problems that can be solved by polytime algorithms. For example, Shortest Paths, MST, MCA $\in \boldsymbol{P}$. Other YES/NO problems:

- `IsComposite`: Given integer $n \geq 2$, is n a composite number; i.e, do exist integers $x, y \geq 2$ such that $n = xy$?
- `Factoring`: Given integer $n \geq 2$, decide if n is composite and if so, find integers $x, y \geq 2$ such that $n = xy$.
- `IsPrime`: Given integer $n \geq 2$, is n a prime number?

Algorithm 8: Factoring algorithm: Sieve of Eratosthenes

1 Consider all integers x such that $2 \leq x \leq \sqrt{n}$, and see if x divides n.
2 If so, then n is composite and $x, \frac{n}{x}$ is a factorization of n.

Is the above a polytime algorithm? No, since number of iterations $\approx \sqrt{n}$, which is not polynomial in $O(\log n)$ which is the input size.

Open question: Is `Factoring` $\in \boldsymbol{P}$? We already know that `IsPrime` $\in \boldsymbol{P}$, which is only settled in around 2003. This implies `IsComposite` $\in \boldsymbol{P}$.

Decision problem is problem with a YES/NO answer. We can take any optimization problem and consider a decision-version of the problem:

- Is there a solution of cost $\leq k$ (for minimization problems)
- Is there a solution of value $\geq k$ (for maximization problems)

Here is decision version of MST (`DecMST`): given $(G, \{c_e\})$, and a number k, is there a spanning tree of cost $\leq k$? If MST $\in \boldsymbol{P}$, then `DecMST` $\in \boldsymbol{P}$.

5.4 Polynomial-Time Reductions

polytime reduction

Given problems A, B, we say ***A* reduces in polynomial time to *B***, or ***A* is polytime reducible to *B***, denote $A \leq_p B$, if we can solve problem A using a polynomial number of elementary operations + a polynomial number of calls to an algorithm for problem B.

$MST \leq_p MCA$. Given input ($G = (V, E)$ undirected graph, $\{c_e\}_{e \in E}$) to MST problem, create $(\overleftrightarrow{G}, \overleftrightarrow{c})$ the bidirected version of G, by creating edges (u, v), (v, u) in $\overleftrightarrow{G}$ for every edge uv of G, and give these two edges cost c_{uv}. Solve MCA on $(\overleftrightarrow{G}, \overleftrightarrow{c}, r)$ where r is any node of V, and map back MCA to undirected edge of G to get an MST.

$MST \leq_p$ Min Steiner Tree Problem. This is true because MST is a special case of Min Steiner Problem, when edge costs are ≥ 0. For MST, can always ensure (i.e., move to) nonnegative edge costs by adding a suitable large constant to all edge costs.

Suppose $A \leq_P B$. Then

1. If $B \in \boldsymbol{P}$, then $A \in \boldsymbol{P}$.
2. Equivalently, if $A \notin \boldsymbol{P}$, then $B \notin \boldsymbol{P}$.

Consider

- `SORTING`: Given n numbers $a_1, \ldots, a_n$, sort them in an increasing order.
- `MIN`: Given n numbers $a_1, \ldots, a_n$, find the minimum of these n numbers.

`SORTING` $\leq_p$ `MIN`: Can sort using n calls to an algorithm for `MIN`. Also `MIN` $\leq_p$ `SORTING`.

Consider [2]

- `LPFeas`: Given $A \in \mathbb{Q}^{m \times n}$, is the system $Ax \leq b, x \geq 0$ feasible?
- `LPOpt`: Given $A \in \mathbb{Q}^{m \times n}$, $b \in \mathbb{Q}^m$, $c \in \mathbb{Q}^n$, does the LP

$$\begin{aligned} \max \quad & c^T x \\ \text{s.t.} \quad & Ax \leq b \\ & x \geq 0 \end{aligned}$$

  have an optimal solution?

`LPFeas` $\leq_p$ `LPOpt`. We can set $c = 0$, and call algorithm for `LPOpt`. Using duality we can show `LPOpt` $\leq_p$ `LPFeas`.

Decision problem `LPImprov`: Given $A \in \mathbb{Q}^{m \times n}$, $b \in \mathbb{Q}^m$, $c \in \mathbb{Q}^n$, and $z \in \mathbb{Q}$, determine if there is a feasible solution to the LP:

$$\begin{aligned} \max \quad & c^T x \\ \text{s.t.} \quad & Ax \leq b \\ & x \geq 0 \end{aligned}$$

with objective value $> z$? Using duality, can show that `LPImprov` $\leq_p$ `LPFeas`.

Note that simplex method for solving LPs is NOT a polytime algorithm.

Consider

- `MST`: Given connected graph $G = (V, E)$, edge costs $\{c_e\}$ that are positive integers, find cost of MST.
- Decision version of MST, `DecMST`: Given connected $G = (V, E)$, positive integer edge costs $\{c_e\}$, and an integer $k > 0$, is there a spanning tree of cost $\leq k$?

We observe that `DecMST` $\leq_P$ `MST`, `MST` $\leq_P$ `DecMST`.

Given MST instance (G, c), we can deduce that every spanning tree has cost $\geq LB := 1$, and MST has cost $\leq UB := (n-1)c_{\max}$, where $c_{\max} := \max_e c_e$.

[2] Here we choose to work with rational numbers because it's hard to specify the input size for irrational numbers

Algorithm Start at $k = UB$. Solve `DecMST` with this k; if answer is YES, set $k \leftarrow k - 1$ and REPEAT.

This is not a polytime reduction, since we could be calling algorithm for `DecMST` $\approx UB$ times, but $UB \approx (n-1)c_{\max}$ is not polynomially bounded in input size.

We can get polynomial reduction by doing binary search. Binary search calls algorithm `DecMST` $O(\log_2 UB)$ which is polynomially bounded in input size.

It's not hard to see that $\leq_p$ is transitive.

Consider two problems:

- MST: find cost of MST
- A-MST: Find a min-cost spanning tree.

We can see that MST $\leq_p$ A-MST. A-MST $\leq_p$ MST: Let $k = MST(G, c)$. Let e be some edge of G. Then $k' = MST(G - e, \{c_f\}_{f \in G-e})$. If $k' > k$, then e belongs to every MST. If $k' \leq k$, then $G \leftarrow G - e$, $c \leftarrow \{c_f\}_{f \neq e}$.

6

P & NP

Consider decision version of Steiner tree:

`DecSteiner`: Given an instance (G, c, T) and a number k, it there a Steiner tree of cost $\leq k$?

Observation Is `DecSteiner` instance is a YES instance, then there is a simple certificate (A steiner tree of cost $\leq k$) to convince one that answer is indeed YES.

verifier/certifier

A verifier/certifier V for a decision problem Π is an algorithm that takes two inputs, x: an instance of Π, and a "certificate" y, and outputs YES or NO. It satisfies

- If x is a YES instance, then there exists y such that $V(x, y) =$ YES.
- If x is a NO instance, then for all y, $V(x, y) =$ NO.

6.1 NP

NP a decision problem Π is in the class ***NP*** if there exists a polynomial p and a polytime verifier V for Π such that

- For every YES instance x of Π, there exists a certificate y with $\text{size}(y) \leq p(\text{size}(x))$ such that $V(x, y) =$ YES.
- (For every NO instance x, $V(x, y) =$ NO for all y)

Informally, a decision problem is in ***NP*** if its YES instances have "short", efficiently verifiable certificates.

> Example:
>
> `DecSteiner` $\in$ ***NP***. It has certificate y: Steiner tree of cost $\leq k$. Verifier V: check y is indeed a Steiner tree of cost $\leq k$.

> Example:
>
> Recall `IsComposite`. It is in ***NP***. Certificate y: a factor of n where $2 \leq y < n$. Verifier: check y divides n (and $2 \leq y < n$).

Example:
`IsPrime` $\in$ ***NP***? Yes, certificate for YES instance uses results in number theory, Fermat's little theorem. We know that `IsPrime` $\in$ ***P***.

Claim 6.1

If Π is a decision problem in ***P***, then Π is also ***NP***. In other words, ***P*** $\subseteq$ ***NP***.

Proof:
Let A: polytime algorithm for Π. Then, we can construct a polytime verifier V for Π as follows: on input (x, y), V returns $A(x)$. □

Exercise:
Prove that ***P*** $\neq$ ***NP***.

Example: `3-SAT`
Given a boolean formula F of the form

$$F = C_1 \wedge C_2 \wedge \cdots \wedge C_m,$$

where each C_i is a clause of the form $y_{i_1} \vee y_{i_2} \vee y_{i_3}$ where each y_i (called literal) is x_i or $\bar{x}_i$.

Is there are True/False assignment of the variable under which F evaluates to T? Such an assignment is called a satisfying assignment.

Not hard to see `3-SAT` $\in$ ***NP***: certificate y: satisfying assignment. Verifier: check that y is indeed a satisfying.

Example: `Set-Cover`
Input: a universe U, and a collection $\mathcal{S} = \{S_1, \ldots, S_m\}$ of subsets of U, and an integer $k \geq 0$.

Are there k subsets $S_{i_1}, \ldots, S_{i_k} \in \mathcal{S}$ such that $(S_{i_1} \cup \cdots \cup S_{i_k}) = U$?

`Set-Cover` $\in$ ***NP***: a certificate for YES instance is simply k sets of $\mathcal{S}$ whose union is U.

6.2 NP-hard and NP-complete

NP-hard, NP-complete

A problem B is called:

- NP-hard: if $x \leq_p B \ \forall x \in$ ***NP***
- NP-complete: if $B \in$ ***NP*** and B is NP-hard (i.e., $x \leq_p B \ \forall x \in$ ***NP***)

Intuitively, NPC are hardest problems in class ***NP***.

Suppose Y is a NPC problem. Then

- if $Y \in$ ***P***, then ***P*** = ***NP***. Since $X \leq_p Y$ for all $X \in$ ***NP*** and $Y \in$ ***P***, we get that $x \in$ ***P*** $\forall x \in$ ***NP***.
- If $Y \notin$ ***P***, then ***P*** $\neq$ ***NP***.

How do we show that a problem Y is NP-hard or NPC?

- Pick a suitable NP-hard problem B.
- Show $B \leq_P Y$. Then Y is NP-hard by transitivity of $\leq_p$.

- If we want to show that Y is NP-complete, then also SHOW $Y \in \boldsymbol{NP}$, which is usually easy.

> **Note:**
> If Y' is the decision version of optimization problem Y and Y' is NP-hard, then Y is also NP-hard. Since $Y' \leq_p Y$.

7

NP-Hard and NP-Complete

7.1 Cook-Levin Theorem

Theorem 7.1: Cook-Levin Theorem

3-SAT is NP-complete.

Will show $\Pi_{set\ cover}$ is NPC by showing that 3-SAT $\le_p$ Set Cover. (Already know set cover $\in \boldsymbol{NP}$)

Will show that 3-SAT $\le_p$ special case of set cover, where we have an undirected graph $G = (V, E)$, and that defines

- $U = E$
- every vertex $v \in V$ creates a set in $\mathcal{S}$ given by $\delta(v)$.

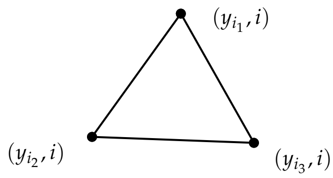
So a collection of sets $\mathcal{S}' = \{\delta(v) : v \in T\}$ is a set cover. Here T is called vertex cover.

Theorem 7.2

3-SAT $\le_p$ Vertex Cover.

Proof:
Let $F = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ where each $C_i = y_{i1} \vee y_{i2} \vee y_{i3}$ be an instance of 3-SAT. Will create a VC-instance such that F is satisfiable $\Leftrightarrow$ VC instance is a YES instance.

- For every clause $C_i = y_{i_1} \vee y_{i_2} \vee y_{i_3}$, create the triangle,

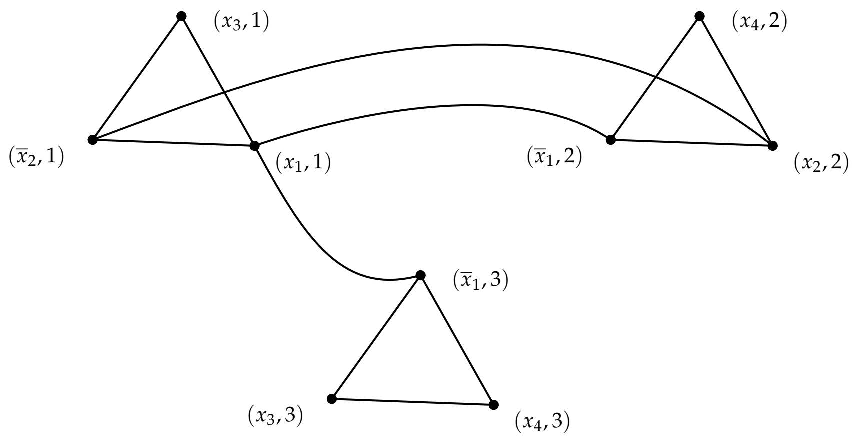


- For every var x_j if $x_j \in C_i$, and $\bar{x}_j \in C_k$, then we create edge $(x_j, i) - (\bar{x}_j, k)$.

So graph $G = (V, E)$ has

$$V = \{(y_j, k) : y_j \in C_k\} \qquad E = \{(x_j, i)(x_{j'}, k) : i = k, \text{ or } x_{j'} = \bar{x}_j\}$$

For example, let $F = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_4 \vee x_2) \wedge (x_3 \vee x_4 \vee \bar{x}_1)$.



We want to show that F is satisfiable $\Leftrightarrow$ the graph created has a VC of size (number of nodes in VC solution) $2m$ where m is number of clauses in F.

Suppose F satisfiable. Then for every C_i, at least one literal in C_i is set to True. Pick exactly one literal, say $y_i^* \in C_i$, that is set to True and consider set $S = \{(y_i, i) : y_i \neq y_i^*\}$. So $|S| = 2m$. We claim that S is a vertex cover.

Clearly all $\triangle$-edges are covered since we pick 2 nodes from each $\triangle$. Consider an edge $(x_j, i)(\bar{x}_j, k)$. It cannot be that both $x_j, \bar{x}_j$ are true, so at least one of these has not been picked for the corresponding clauses (as y_i^*), and so is in S.

Conversely, **suppose the graph has been created has a VC S with $|S| \leq 2m$**. Note that $|S| = 2m$, since S must contain ≥ 2 nodes from each triangle, therefore S contains exactly 2 nodes from C_i's clause-$\triangle$.

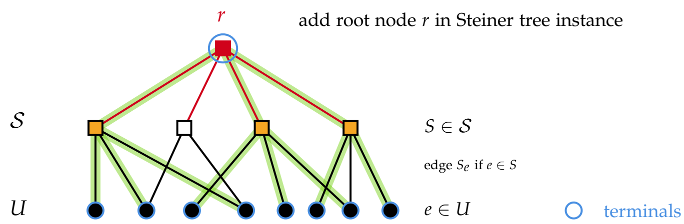
Suppose for C_i, S contains nodes $(y_{i_1}, i), (y_{i_2}, i)$. Then we set $y_{i_3} =$ true. We claim that this gives a satisfying (possibly partial) assignment. Clearly, every clause is set to be true. Consider some var x_j and suppose $(x_j, i)(\bar{x}_j, k) \in V$. Then at least one of these nodes is in S, so we would not set both $x_j, \bar{x}_j$ to true, which is then a valid assignment as this holds for every variable. □

Theorem 7.3

`Set Cover` $\leq_p$ `Steiner Tree`.

These are decision versions of the problems. Hence, decision version of Steiner tree is NP-complete.

Proof:
Let $(U, \mathcal{S} \subseteq 2^U, k)$ be a set cover instance. We represent SC instance $(U, \mathcal{S})$ as follows:



We create Steiner tree instance by appending bipartite graph representing $(U, \mathcal{S})$ with a "root" node r, and edges rS $\forall S \in \mathcal{S}$. Terminal set $T = \{r\} \cup U$. Set edge cost

$$c_f = \begin{cases} 1 & f \in \delta(r) \\ k+1 & f \text{ is incident to an element of } U \end{cases}$$

Then we want to show $\exists$ set cover of size $\leq k \Leftrightarrow \exists$ Steiner tree of cost $\leq \underbrace{n(k+1)+k}_{M}$ where $n = |U|$.

Suppose $\mathcal{S}' \subseteq \mathcal{S}$ is a collection of $\leq k$ sets whose union is U, the tree with edges $\{rS_i : S_i \in \mathcal{S}'\}$ and $\{eS_j : S_j$ in some set in $\mathcal{S}'$ containing $e\}$ is a Steiner tree of cost $\leq n(k+1)+k$.

Suppose F is a Steiner tree of cost $\leq M$. Then every node $e \in U$ must be a leaf node in F, otherwise $c(F) \geq (k+1)(\#$ edges in F incident to nodes in $U) \geq (n+1)(k+1) > M$. Then there are n edges joining edges $e \in U$ to set nodes. Then there are $\leq k$ edges from $\delta(r)$ are in F.

Consider $\mathcal{S}' = \{S_i : rS_i \in F\}$. Then $|\mathcal{S}'| \leq k$ and every $e \in U$ is connected to r by a 2-hop path e-S_i-r, and so $S_i \in \mathcal{S}'$. Therefore $\mathcal{S}'$ is a set cover. $\square$

8

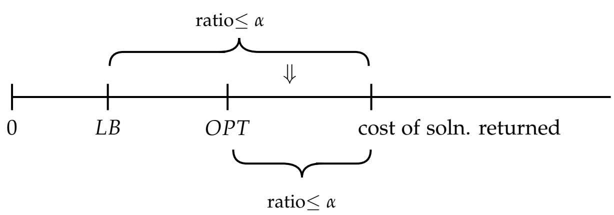
Approximation Algorithm & Primal Dual Algorithm

8.1 Approximation Algorithm Design

Recall the definition for α-approximation algorithm where $\alpha \geq 1$.

Strategy for approximation algorithm design

1. Come up with a (move tractable) lower bound LB on OPT ($LB \leq OPT$)
2. Come up with a polytime algorithm and show that it returns a solution of cost $\leq \alpha \cdot LB \leq \alpha \cdot OPT$.



How to come up with lower bounds? We can formulate discrete optimization problem as an integer program, and relax integrality constraints to get linear program. Then $OPT_{LP} \leq OPT_{IP}$ and now OPT_{LP} is LB.

> Example: Steiner tree
>
> Instance $(G = (V, E), T \subseteq V, \{c_e \geq 0\}_{e \in E})$
>
> Variables: $x_e \in \{0, 1\}$ $\forall e \in E$, and $x_e = 1$ indicates e is in opt tree; $x_e = 0$ indicates e is NOT in our tree. Here x_e is binary indicator variables.
>
> Then the IP:
>
> $$\begin{aligned} \min \quad & \textstyle\sum_{e \in E} c_e x_e \\ \text{s.t.} \quad & x_e \in \{0,1\} \qquad \forall e \in E \\ & \textstyle\sum_{e \in \delta(S)} x_e \geq 1 \quad \forall S \subseteq V : S \cap T \neq \varnothing, T - S \neq \varnothing \end{aligned} \tag{IP}$$
>
> The second condition means for every cut $(S, V - S)$ where S and S^C contain some terminals, then the boundary of the cut must include at least one edge from the tree, otherwise tree might not connect up all terminal nodes.
>
> We then can relax first integrality constraint to $0 \leq x_e \leq 1$, or even further $x_e \geq 0$ because all

feasible $x_e > 1$ can be "reduced to" $x_e = 1$, which still satisfies the constraints. Then LP is

$$\begin{array}{lll} \min & \sum_{e\in E} c_e x_e & \\ \text{s.t.} & x_e \geq 0 & \forall e \in E \\ & \sum_{e\in\delta(S)} x_e \geq 1 & \forall S \subseteq V : S\cap T \neq \varnothing, T - S \neq \varnothing \end{array} \tag{LP}$$

Then $OPT_{LP} \leq OPT_{IP}$.

LP relaxation will be exploited in two ways for designing algorithms

1. LP-rounding algorithms: Solve LP-relaxation to get a (likely) fractional solution x^*. Devise a polytime algorithm to transform x^* to an integer solution $\hat{x}$ and show that

$$cost(\hat{x}) \leq \alpha \cdot cost(x^*) = \alpha \cdot OPT_{LP} \leq \alpha \cdot OPT$$

And this way of transforming is known as LP-rounding algorithm.

Theorem 8.1

LPs can be solved in polytime. (BUT simplex is NOT polytime)

Even there are exponentially many constraints, we can still solve LP in polytime via ellipsoid. Interested readers can consult CO 255 or CO 471.

2. Primal-Dual Algorithms. Recall dual of (LP) is a maximization LP (D), and every dual feasible solution y has value at most OPT_{LP}. So if we design an algorithm that constructs an integer feasible solution $\hat{x}$ and a dual solution y, and show that $cost(\hat{x}) \leq \alpha \cdot (value(y))$. Then we get an α-approximation algorithm since $value(y) \leq OPT_{LP} \leq OPT$.

Recap of LP duality It's bad to memorize the table as said by the instructor. It's better to understand the logic of taking dual.

Back to Steiner tree...

$$\begin{array}{lll} \min & \sum_{e\in E} c_e x_e & \\ \text{s.t.} & x \geq 0 & \\ & \sum_{e\in\delta(S)} x_e \geq 1 & \forall S \in \mathcal{S} \end{array} \tag{P}$$

where $\mathcal{S} = \{S \subseteq V : S \cap T \neq \varnothing, T - S \neq \varnothing\}$.

Now consider the dual constraint:

$$\sum_{e\in E} x_e \Big(\sum_{\substack{S\in\mathcal{S}: \\ e\in\delta(S)}} y_S \Big) = \sum_{S\in\mathcal{S}} y_S \Big(\sum_{e\in\delta(S)} x_e \Big) \geq \sum_{S\in\mathcal{S}} y_S \tag{**}$$

To ensure $\sum_{e\in E} c_e x_e \geq$ LHS of (**), we need that $c_e \geq \sum_{S\in\mathcal{S}:e\in\delta(S)} y_S$. So the dual of (P) is

$$\begin{array}{lll} \max & \sum_{S\in\mathcal{S}} y_S & \\ \text{s.t.} & \sum_{S\in\mathcal{S}:e\in\delta(S)} y_S \leq c_e & \forall e \in E \\ & y \geq 0 & \end{array} \tag{D}$$

Weak Duality

The value of any feasible solution y to (D) provides a lower bound on OPT(P) (when (P) is minimization LP).

So if x is feasible solution to (P), y feasible to (D), then $\sum_{e\in E} c_e x_e \geq \sum_{S\in\mathcal{S}} y_S$.

Strong Duality

If (P) and (D) are both feasible, then they both have optimal solutions, and OPT(P) = OPT(D).

So $\sum_{e \in E} c_e x_e^* = \sum_{S \in \mathcal{S}} y_S^*$.

x^* and y^* optimal $\Leftrightarrow$

- x^*, y^* feasible, AND
- x^*, y^* satisfy the following Complementary Slackness conditions:
    1. $x_e^* > 0 \Rightarrow \sum_{S \in \mathcal{S}: e \in \delta(S)} y_S^* = c_e$
    2. $y_S^* > 0 \Rightarrow \sum_{e \in \delta(S)} x_e^* = 1$

8.2 Network Connectivity Problems

Modeling Framework

We will model a network connectivity problem by a cut-requirement function $f : 2^V \to \{0, 1\}$ where $f(S) = 1$ for $S \subseteq V$ indicates that a feasible solution and must include an edge from $\delta(S)$. This gives rise to the following ***f*-connectivity problem**: Find a min-cost set of edges F s.t. $F \cap \delta(S) \neq \varnothing$ $\forall S \subseteq V$ s.t. $f(S) = 1$.

Example: Steiner tree

Steiner tree is modeled by the cut-requirement function f^{ST} where

$$f^{ST}(S) = \begin{cases} 1 & \text{if } S \cap T \neq \varnothing, T - S \neq \varnothing \\ 0 & \text{otherwise} \end{cases}$$

Then Steiner tree is equivalent to f^{ST}-connectivity problem. I.e., F a Steiner tree $\Leftrightarrow$ F is an acyclic solution to f^{ST}-connectivity.

Note that $f(S) = 0$ does not say anything about $F \cap \delta(S)$.

8.3 LP-relaxation for f-connectivity and its dual

Define $\mathcal{S} = \{S \subseteq V : f(S) = 1\}$.

$$\begin{array}{lll} \min & \sum_{e \in E} c_e x_e & \\ \text{s.t.} & \sum_{e \in \delta(S)} x_e \geq 1 & \forall S \in \mathcal{S} \\ & x \geq 0 & \end{array} \tag{P}$$

$$\begin{array}{lll} \max & \sum_{S \in \mathcal{S}} y_S & \\ \text{s.t.} & \sum_{S \in \mathcal{S}: e \in \delta(S)} y_S \leq c_E & \forall e \in E \\ & y \geq 0 & \end{array} \tag{D}$$

We will consider f-connectivity for a structured class of cut-requirement functions.

$\{0,1\}$-proper cut-requirement function

A $\{0,1\}$-proper cut-requirement function is a function $f : 2^V \to \{0,1\}$ such that

(i) $f(V) = 0$;

(ii) $f(S) = f(V - S)$ $\forall S \subseteq V$; (symmetry)

(iii) for any $A, B \subseteq V$ s.t. $A, B \neq \varnothing$, $A \cap B = \varnothing$, $f(A \cup B) = 1 \Rightarrow f(A) = 1$ or [a] $f(B) = 1$.

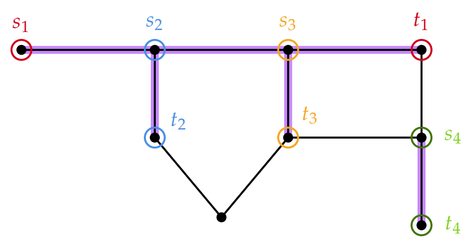
[a] non-exclusive, could have $f(A) = f(B) = 1$

We observe that f^{ST} is a $\{0,1\}$-proper function.

Example: Generalized Steiner tree/Steiner forest

Given undirected graph $G = (V, E)$, edge costs $\{c_e \geq 0\}$, k-disjoint terminal sets $T_1, \ldots, T_k \subseteq V$, find a min-cost set of edges F s.t. each T_i, $i = 1, \ldots, k$, belongs to a connected components of (V, F) in T_i. I.e., all nodes are connected in F.

For example,



where four colors represent four T_i's, and the purple forest is a feasible solution.

This can be modelled by the proper function.

$$f^{GST}(S) = \begin{cases} 1 & \text{if } \exists i \text{ such that } S \cap T_i \neq \varnothing, T_i - S \neq \varnothing \\ 0 & \text{otherwise} \end{cases}$$

It's not hard to show that Generalized Steiner tree is equivalent to f^{GST}-connectivity: write one T_i, Generalized Steiner tree $\to$ Steiner tree, $f^{GST} \to f^{ST}$.

We claim that f^{GST} is a proper function.

Proof:

Clearly $f^{GST}(V) = 0$ and symmetry holds. For (iii), suppose $A, B \neq \varnothing, A \cap B = \varnothing$, s.t. $f^{GST}(A \cup B) = 1$. So there exists i such that $(A \cup B) \cap T_i \neq \varnothing$, $T_i - (A \cup B) \neq \varnothing$. I.e.,

(a) $A \cap T_i \neq \varnothing$ (i.e., $T_i - A \neq \varnothing$) OR

(b) $B \cap T_i \neq \varnothing$ (i.e., $T_i - B \neq \varnothing$).

So $f^{GST}(A) = 1$ (a) OR $f^{GST}(B) = 1$ (b). □

Goal Design a 2-approximation algorithm for f-connectivity with $\{0,1\}$-proper function.

Recall (P) and (D) defined previously. **Primal-Dual Algorithm** will simultaneously construct an integer feasible solution to (P), and a dual feasible solution y.

It's useful to introduce a notion of "time" t. Initially $t = 0$ and t increases at rate 1. (t is not necessarily an integer) We will maintain a set F of edges, and a dual feasible solution y. Initially $F \leftarrow \varnothing$, $y_S \leftarrow 0$ $\forall S \in \mathcal{S}$, which is feasible. At any state, given *infeasible* solution F, there will be some **violated sets**, i.e., set S with $f(S) = 1$ BUT $F \cap \delta(S) = \emptyset$. We will increase y_S for some violated sets.

Which violated sets do we pick?

Design rule 1 Will pick the (inclusion-wise) minimal violated sets (MVSs). Let

$$\mathcal{V} = \{S \in \mathcal{S} : S \text{ is violated, and } S \text{ is minimal among violated sets}\}$$

"Minimal among violated sets": $\forall T \subsetneq S$, T is not violated.

Now given $\mathcal{V}$, we want increase y_S $\forall S \in \mathcal{V}$.

Lemma 8.2

Given set F of edges, the MVSs are $\{S \subseteq V : S \text{ is a component of } (V, F) \text{ with } f(S) = 1\}$.

Proof:
Clearly if $F \cap \delta(S) \neq \emptyset$, then S is not violated. So any violated set S must be a union $C_1 \cup \cdots \cup C_k$ of components of (V, F). Since $f(S) = 1$, by property (iii) of proper functions, we must have $f(C_i) = 1$ for some $i = 1, \ldots, k$. Thus C_i is a MVS. □

9

Primal Dual Algorithm cont'd

9.1 Primal dual algorithm for f-connectivity

Recall $\mathcal{S} = \{S \subseteq V : f(S) = 1\}$.

$$\begin{array}{lll} \min & \sum_{e\in E} c_e x_e & \\ \text{s.t.} & \sum_{e\in\delta(S)} x_e \geq 1 & \forall S \in \mathcal{S} \\ & x \geq 0 & \end{array} \tag{P}$$

$$\begin{array}{lll} \max & \sum_{S\in\mathcal{S}} y_S & \\ \text{s.t.} & \sum_{S\in\mathcal{S}: e\in\delta(S)} y_S \leq c_E & \forall e \in E \\ & y \geq 0 & \end{array} \tag{D}$$

Corollary 9.1

$F \subseteq E$ is feasible (for f-connected with a $\{0,1\}$-proper function) $\Leftrightarrow$ $f(S) = 0$ for every component S of (V, F).

Design Rule 2 Increase y_S $\forall S \subseteq \mathcal{V}$ uniformly at rate 1, (but y_S's need not be integral) until constraint $\sum_{S\in\mathcal{S}:e\in\delta(S)} y_S \leq c_E$ goes tight for some edge $e \in \delta(S)$ for some MVS S. When this happens, we add e to F.

We STOP, when F is feasible, i.e., there are no violated sets (and hence no MVSs), where the feasibility of F can be detected using Corollary 9.1.

Final Design Rule (Reverse Delete) After we have stopped adding edges, consider edges in F in reverse insertion order and delete an edge e if $F - \{e\}$ is feasible.

Algorithm 9: Primal Dual Algorithm

Initialize $F \leftarrow \varnothing$, $y_S \leftarrow 0\ \forall S \in \mathcal{S}$, and time $t \leftarrow 0$ (t increase at rate 1 throughout).
Let $\mathcal{V}$: MVSs give F, i.e., $\{S \subseteq V : S$ is a component of (V,F) with $f(S) = 1\}$. `// Lemma 8.2`
while $\mathcal{V} \neq \varnothing$ **do**
  (a) Increase y_S uniformly at rate 1 $\forall S \in \mathcal{V}$ until some edge $e \in \delta(S)$ goes tight for some $S \in \mathcal{V}$.
  (b) Update $F \leftarrow F \cup \{e\}$, update $\mathcal{V}$.
(Reverse Delete) Let $F = \{e_k, e_{k-1}, \dots, e_1\}$ where e_k is inserted last, e_{k-1} before e_k and so on. For $i = k..1$, if $F - \{e_i\}$ is feasible, $F \leftarrow F - \{e_i\}$.
return F

Example:

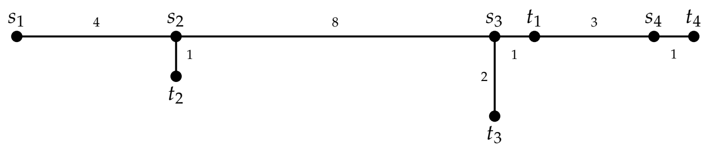
Consider Steiner tree with $T = \{s,t\}$, s-t shortest path. At each point of time, we have exactly 2 MVSs, consisting of components of (V,F) are containing s, other containing t. Then Primal Dual algorithm is equivalent to bidirectional version of Dijkstra, where we grow explored set simultaneously from both s,t.

Example:

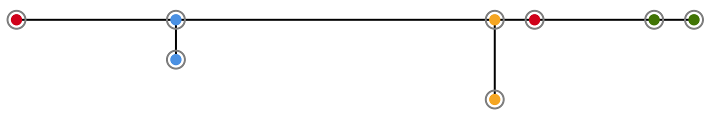
Consider Steiner tree with $T = V$, i.e., MST. Then Primal Dual algorithm is exactly Kruskal's algorithm.

First edge to tight is least-cost edge. In general, given F, next edge to go tight in the least cost edge joining 2 components of (V,F). This is because any edge e that gets added to F by Primal Dual algorithm, is on the boundary of exactly 2 sets in V at all times until it is added. Thus e gets added at time $t_e = c_e/2$.

Now consider the example in Generalized Steiner Tree. Let $T_i = \{s_i, t_i\}$ for $i = 1,2,3,4$.



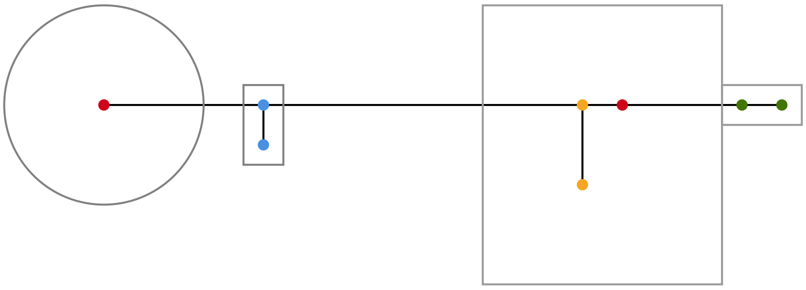
Initially, all singleton nodes are violated sets as $f(\{v\}) = 1$ and it requires an edge in $\delta(\{v\})$ for all $v \in \cup T_i$.



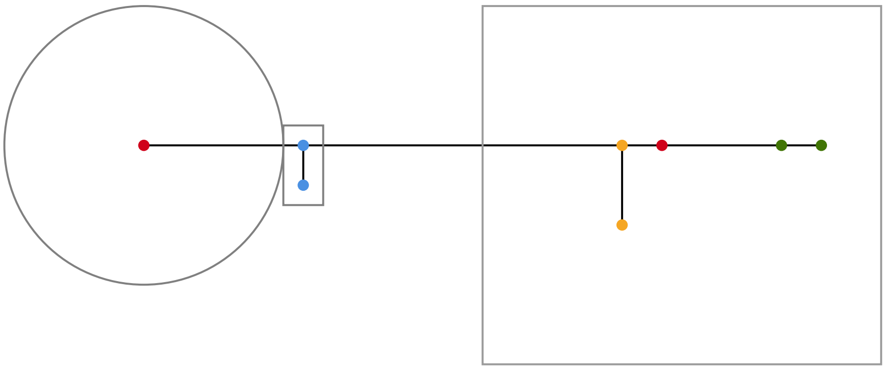
$t = 0$, $y_S \leftarrow 0\ \forall S \in \mathcal{S}$, $F \leftarrow \varnothing$.

$t = 0.5$, $y_{\{v\}} = 0.5\ \forall v \in \cup T_i$. $F \leftarrow \{s_4t_4, s_3t_1, s_2t_2\}$. $\mathcal{V} \leftarrow \left\{\underset{0.5}{\{s_1\}}, \underset{0}{\{s_3, t_1\}}, \underset{0.5}{\{t_3\}}\right\}$. The red values are y_S. $F = \{s_2t_2, s_3t_1, s_4t_4\}$.

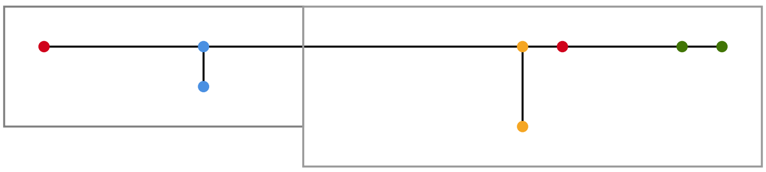
$t = 1$, s_3t_3 goes tight. $y_{\{s_3,t_1\}} = 0.5$, $y_{\{s_1\}} = y_{\{t_3\}} = 1$, $y_{\{v\}} = 0.5\ \forall v \in \cup T_i - \{s_1, t_3\}$.
Here $\mathcal{V} = \left\{ \underset{1}{\{s_1\}}, \underset{0}{\{s_3t_1t_3\}} \right\}$. $F = \{s_2t_2, s_3t_1, s_4t_4, s_3t_3\}$.



$t = 2.5$, t_1s_4 goes tight. Here $\mathcal{V} = \left\{ \underset{2.5}{\{s_1\}}, \underset{0}{\{s_3s_4t_1t_3t_4\}} \right\}$. $F = \{s_2t_2, s_3t_1, s_4t_4, s_3t_3, t_1s_4\}$.



$t = 3.5$, s_1s_2 goes tight. $\mathcal{V} = \left\{ \underset{0}{\{s_1s_2t_2\}}, \underset{1}{\{s_3s_4t_1t_3t_4\}} \right\}$. $F = \{s_2t_2, s_3t_1, s_4t_4, s_3t_3, t_1s_4, s_1s_2\}$.



$t = 5.5$, s_2s_3 goes tight. Final dual solution is

$y_{s_1} = 3.5$	$y_{t_1} = 0.5$	$y_{s_2t_2} = 0$	$y_{s_1s_2t_2} = 2$
$y_{s_2} = 0.5$	$y_{t_2} = 0.5$	$y_{s_3t_1} = 0.5$	$y_{s_3s_4t_1t_3t_4} = 3$
$y_{s_3} = 0.5$	$y_{t_3} = 1$	$y_{s_4t_4} = 0$	
$y_{s_4} = 0.5$	$y_{t_4} = 0.5$	$y_{s_3t_1t_3} = 1.5$	

And there's no violated sets!

Then the final edge set F is

where $t_1 s_4$ gets deleted.

Here $\{s_2, t_2\}, \{s_4, t_4\}$ are components that are not violated, i.e., $f(C) = 0$.

9.2 Analysis

Now back to the algorithm. We want to analyze **running time**. We only need tto keep track of non-zero y_S values. We can easily identify MVSs, $\leq n$ MVSs any point.

F is always acyclic, since we only add edges on boundaries of some components. [1] Thus $|F| \leq n-1$. Therefore, at most $n-1$ iterations of WHILE loop in step 2. Moreover, as there are at most n MVSs, then there are at most n^2 sets S with $y_S > 0$.

We can compute (in polytime) next edge to go tight (and also the time at which this happens) by keeping track of "residual cost" $c_e - \sum_{S \in \mathcal{S}: e \in \delta(S)}$ for each $e \in E$. This implies WHILE loop runs in polytime, and Reverse Delete takes polytime.

Now let's examine the **approximation guarantee**. Recall the CS conditions for (P), (D):

1. $x_e > 0 \Rightarrow \sum_{S \in \mathcal{S}: e \in \delta(S)} y_S = c_e \ \forall e \in E$. This holds by design $\forall e \in F$.
2. $y_S > 0 \Rightarrow \sum_{e \in \delta(S)} x_e = 1 \ \forall S \in \mathcal{S}$. This can be violated.

Exercise:
If (1) holds, and the following relaxation of (2) holds: (2') $y_S > 0 \Rightarrow \sum_{e \in \delta(S)} x_e \leq 2$ (where x is $\{0,1\}$-solution corresponding to F: $x_e = 1 \ \forall e \in F$, $x_e = 0$ otherwise).

Then $\sum_e c_e x_e \leq 2 \sum_{S \in \mathcal{S}} y_S$, which is then a 2-approximation.

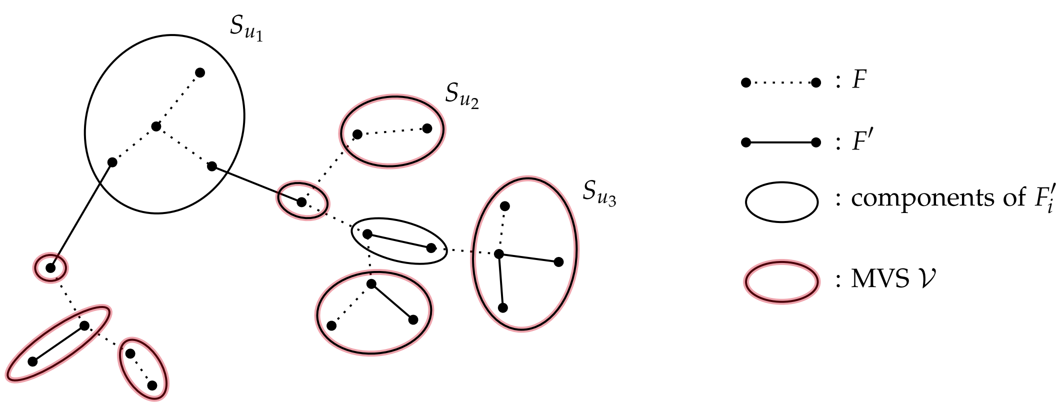
Notation For an edge set $Z \subseteq E$, $\delta_Z(S) = \delta(S) \cap Z$. From now on, F is the **final-set** of edges returned.

Lemma 9.2

At any point in the algorithm, $\sum_{S \in \mathcal{V}} |\delta_F(S)| \leq 2|\mathcal{V}|$.

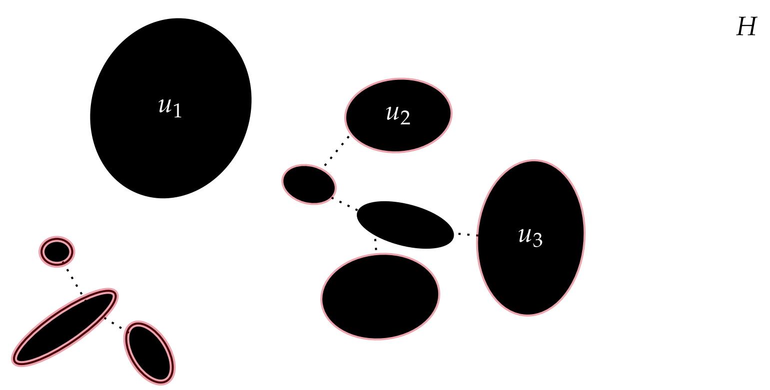
This is like an "averaged" version of (2').

Proof:
Recall F is final edge set. Consider (start of) some stage i. Let $F' \supseteq F$ be the edge-set BEFORE Reverse Delete. Let $F'_i \subseteq F'$ be edges added to F' by START of stage i. Let $\mathcal{V} = \mathcal{V}_i$: collection of MVSs at start of stage i.



[1] A tiny note: we might encounter the situation that at some point, two edges connecting two components go tight at the same time. We are processing them in linear order. After we add one edge, then the other edge is no longer on the boundary, so the other edge is not added.

Let H be graph obtained from (V, F) by contracting the node-sets corresponding to components of (V, F_i') where we eliminate self loops.



For each node u in H, let $S_u \subseteq V$ be nodes that were contracted to form u. So $\delta_F(S_v) = \delta_H(v)$. We have to show

$$\sum_{v \in H : S_v \in \mathcal{V}} |\delta_H(v)| \leq 2|\{v \in H : S_v \in \mathcal{V}\}|$$

We claim that H is acyclic.

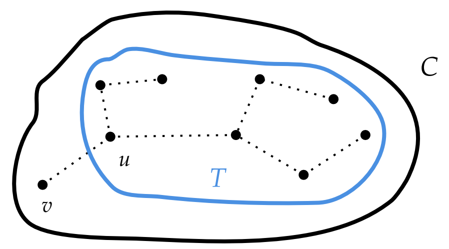
Consider H' be graph obtained from (V, F') by contracting components of (V, F_i'). Since $F', F_i' \subseteq F'$ are acyclic, H' is acyclic. But H is a subgraph of H', so H is acyclic.

Consider a component C of H. We will show

$$\sum_{v \in C : S_v \in \mathcal{V}} |\delta_H(v)| \leq 2|\{v \in C : S_v \in \mathcal{V}\}| \tag{$**$}$$

We observe that adding $(**)$ over all components of H gives $\sum_{v \in H : S_v \in \mathcal{V}} |\delta_H(v)| \leq 2|\{v \in H : S_v \in \mathcal{V}\}|$. Note that $\sum_{v \in C} |\delta_H(v)| = 2(|C| - 1)$ because it is a tree and then by handshaking lemma. So to show $(**)$, it suffices to show that if v is a leaf of C, then $S_v \in \mathcal{V}$. (so if $v \in C$, $S_v \notin \mathcal{V}$, then $|\delta_H(v)| \geq 2$.)

If C is a singleton component, $(**)$ clearly holds. Suppose C is not a singleton component, and let v be a leaf node node of C.



Here we also suppose $f(S_v) = 0$, i.e., $S_v \notin \mathcal{V}$. Since edge uv of C incident to v was not deleted, if $Z \subseteq V$ node-set corresponds to $C - \{v\}$ (i.e., $Z = \bigcup_{w \in C - \{v\}} S_w$), then $f(Z) = 1$. [a] Then $f(V - Z) = 1$. But

$$V - Z = \left(V - \bigcup_{w \in C} S_w\right) \cup S_v$$

and $f(V - \bigcup_{w \in C} S_w) = f(\bigcup_{w \in C} S_w) = 0$ since C is a component of H. This contradicts property (iii) of proper functions.

To finish the proof,

$$\begin{aligned}
\sum_{v\in H:S_v\in\mathcal{V}} |\delta_H(v)| &= \sum_{\text{components } C \text{ of } H} \quad \sum_{v\in C:S_v\in\mathcal{V}} |\delta_H(v)| \\
&= \sum_{\text{components } C \text{ of } H} \Big(2|C| - 2 - \sum_{v\in C:S_v\notin\mathcal{V}} \underbrace{|\delta_H(v)|}_{\geq 2}\Big) \\
&\leq \sum_{\text{components } C \text{ of } H} \Big(2|C| - 2 - \sum_{v\in C:S_v\notin\mathcal{V}} 2\Big) \\
&= \sum_{\text{components } C \text{ of } H} (2|C| - 2|v\in C : S_v\notin\mathcal{V}| - 2) \\
&= \sum_{\text{components } C \text{ of } H} (2|v\in C : S_v\in\mathcal{V}| - 2) \\
&\leq \sum_{\text{components } C \text{ of } H} (2|v\in C : S_v\in\mathcal{V}|) \\
&= \sum_{\text{components } C \text{ of } H} \quad \sum_{v\in C:S_v\in\mathcal{V}} 2 \\
&= 2|\{v\in H : S_v\in\mathcal{V}\}|
\end{aligned}$$

[a]We have $f(Z\cup S_v)=0$ as C is a component of H. So there are no edges in final set F from $\delta(Z\cup S_v)$. uv as not deleted because deleting uv will create an infeasible solution, which means some component S of $(V, F-uv)$ has $f(S)=1$. As F is feasible, then such an S must be a new component that gets created when we delete uv: either $S=S_v$ or $S=Z$. As $f(S_v)=0$, then $f(Z)=1$.

Theorem 9.3

The final edge set F, dual solution y returned satisfy $c(F)\leq 2\sum_{S\in\mathcal{S}} y_S$. Thus 2-approximation.

Proof:
Divide the algorithm into stages, where each stage i corresponds to an iteration of WHILE loop, and on associated time interval. Note that in a stage, collection of MVSs does not change. Let

$$\begin{aligned}
\Delta_i &= \text{length of stage } i \text{ (i.e., length of associated time interval)} \\
\mathcal{V}_i &= \text{collection of MVSs at start of stage } i
\end{aligned}$$

Note that $y_S = \sum_{i:S\in\mathcal{V}_i}\Delta_i \quad \forall S\in\mathcal{S}$, which implies

$$\sum_{S\in\mathcal{S}} y_S = \sum_{S\in\mathcal{S}}\Big(\sum_{i:S\in\mathcal{V}_i}\Delta_i\Big) = \sum_i \Delta_i|\mathcal{V}_i|$$

Then we have

$$\begin{aligned}
c(F) = \sum_{e\in F} c_e &= \sum_{e\in F}\Big(\sum_{S\in\mathcal{S}:e\in\delta(S)} y_S\Big) \\
&= \sum_{S\in\mathcal{S}} y_S|\delta_F(S)| \\
&= \sum_{S\in\mathcal{S}}\Big(\sum_{i:S\in\mathcal{V}_i}\Delta_i\Big)|\delta_F(S)| \\
&= \sum_i \Delta_i\Big(\sum_{S\in\mathcal{V}_i}|\delta_F(S)|\Big) \\
&\leq \sum_i \Delta_i\cdot 2|\mathcal{V}_i| && \text{By Lemma 9.2} \\
&= 2\sum_{S\in\mathcal{S}} y_S
\end{aligned}$$

10

Set Cover & Uncapacitated Facility Location

Recall `Set Cover`: Given a universe U of n elements, and a collection $\mathcal{S}$ of subsets of U. Each set $S \in \mathcal{S}$ a weight $w_S \geq 0$. **Goal**: Find a min-weight collection of sets from $\mathcal{S}$ whose union is U, which is called a set cover.

Recall:

- Even when all $w_S = 1$, `Set Cover` is NP-hard.
- Even vertex cover (special case, where sets is equivalent to nodes of a graph, U is edges) is NP-hard.

10.1 LP-relaxation for SET COVER and its dual

As usual, S index sets in $\mathcal{S}$, e index elements in U.

$$\min \sum_S w_S x_S \tag{SC-P}$$

$$\text{s.t.} \sum_{S:e\in S} x_S \geq 1 \qquad \forall e \in U \tag{1}$$

$$x \geq 0$$

where x_S indicates if S is picked.

$$\max \sum_e y_e \tag{SC-D}$$

$$\text{s.t.} \sum_{e\in S} y_e \leq w_S \qquad \forall S \in \mathcal{S} \tag{2}$$

$$y \geq 0$$

A $\{0,1\}$-solution to (SC-P) is precisely a set cover.

CS conditions

$$x_S > 0 \implies \sum_{e\in S} y_e = w_S \qquad \forall S \in \mathcal{S} \tag{3}$$

$$y_e > 0 \implies \sum_{S:e\in S} x_S = 1 \qquad \forall e \in U \tag{4}$$

Let $B = \max_e |\{S \in \mathcal{S} : e \in S\}|$.

Claim For any feasible solution x to (SC-P), where $x_S \leq 1 \, \forall S$, we have $\sum_{S: e \in S} x_S \leq B$ for every $e \in U$. This is simply because there are at most B terms on LHS, each is at most 1.

So if we can construct a $\{0,1\}$-solution $\hat{x}$ feasible for (SC-P), and a dual feasible solution y such that (3) holds, then

$$\sum_S w_S \hat{x}_S = \sum_{S:\hat{x}_S=1} w_S = \sum_{S:\hat{x}_S=1} \Big(\sum_{e \in S} y_e\Big) = \sum_e y_e \Big(\sum_{S:\hat{x}_S=1, e\in S} 1\Big) \leq B \sum_e y_e \leq B \cdot \mathrm{OPT(SC\text{-}D)}$$

which is B-approximation. This suggests following primal-dual B-approximation algorithm.

Algorithm 10: Primal Dual algorithm for `Set Cover`

1 Initially $y \leftarrow 0$, $\mathcal{C} \leftarrow \varnothing$, $N = U - \sum_{S \in \mathcal{C}} S$ `//` $\mathcal{C}$ `is collection of sets picked.` N `is uncovered elements.`
2 **while** $N \neq \varnothing$ **do**
    Pick some $e \in N$
    Increase y_e until (2) goes tight for some set S s.t. $e \in S$
    $\mathcal{C} \leftarrow \mathcal{C} \cup \{S\}$
    Update N
3 **return** $\mathcal{C}$

Theorem 10.1

This primal dual algorithm is a B-approximation algorithm.

For vertex cover, $B = 2$ because each edge has 2 end points. Thus we have a 2-approximation for vertex cover.

Note that B can be very large, so the algorithm would be bad. Here is another algorithm for `Set Cover`.

Algorithm 11: Greedy algorithm for `Set Cover`

1 $y \leftarrow 0$, $\mathcal{C} \leftarrow \varnothing$, $N \leftarrow U$. Initialize "time" $t \leftarrow 0$ `//` t `increase at rate 1 throughout`
2 **while** $N \neq \varnothing$ **do**
    Raise $y_e \, \forall e \in N$ uniformly at rate 1, until some set S is paid by elements in N, i.e., $\sum_{e \in N \cap S} y_e = w_S$.
    $\mathcal{C} \leftarrow \mathcal{C} \cup \{S\}$, $N \leftarrow N - S$.
3 **return** $\mathcal{C}$

We make several observations

- At any time t, if N_t is the set of uncovered elements of time t, we have $y_e = t \, \forall e \in N_t$.
- So the *first* set to be picked is the set with smallest $w_s/|S|$ ratio.

In general, given that we have picked sets in $\mathcal{C}$, and set of uncovered elements N, the next set to be picked is the set $S \notin \mathcal{C}$ with smallest $\dfrac{w_S}{|S \cap N|}$ ratio, which is the spirit of greediness.

Now let's analyze the greedy approximation algorithm. By design, $\sum_{S \in \mathcal{C}} w_S = \sum_e y_e$. Does this suggest an optimal solution? But note that y need not to be feasible for (SC-D). We could have $\sum_{e \in S} y_e > w_S$ since we ignore y_e's for $e \in S - N$.

Let $\Delta = \max_{S \in \mathcal{S}} |S|$.

For an integer $k \geq 1$, we denote

$$H_k := 1 + \frac{1}{2} + \cdots + \frac{1}{k}$$

which is k-th Harmonic number, and the value is roughly $\log k$. We will show $y' = y/H_\Delta$ is a dual feasible solution, where y is the dual solution at end of greedy algorithm. Then this implies Greedy algorithm is a H_Δ-approximation algorithm:

$$\sum_{S \in \mathcal{S}} = \sum_{e \in U} y_e = H_\Delta \cdot \left(\sum_{e \in U} y'_e \right) \leq H_\Delta \cdot \mathrm{OPT(SC\text{-}D)}$$

Lemma 10.2

$\sum_{e \in S} y_e \leq H_\Delta \cdot w_S$ for all $S \in \mathcal{S}$.

Proof:

Consider any set S. Order elements of S: $e_1, \ldots, e_k$, where $k = |S|$, s.t. e_1 is covered before e_2, e_2 before e_3, and so on. How large could y_{e_1} be ?

Let t_i be time when e_i is covered. So $y_{e_i} = t_i$ by design.

At time $t_1 - \epsilon$ ($\epsilon > 0$ small), all $e_1, \ldots, e_k$ are uncovered. So $|S|(t_1 - \epsilon) \leq w_S$, where LHS is total y-contribution from $|S \cap N|$, then $t_1 \leq \frac{w_s}{k} + \epsilon$. As this holds for $\epsilon > 0$, then $t_1 = y_{e_1} \leq w_S/|S|$.

Now consider y_{e_2}. At this time, all of $e_2, \ldots, e_k$ are uncovered. Then $(|S| - 1)(t_2 - \epsilon) \leq w_S$. This holds for all $\epsilon > 0$. Thus $y_{e_2} = t_2 \leq \frac{w_S}{|S|-1}$.

Continuing this way, we get $y_{e_i} = t_i \leq \frac{w_S}{|S|-i+1}$ for all $i = 1, \ldots, k = |S|$. Thus

$$\sum_{e \in S} y_e = \sum_{i=1}^{|S|} y_{e_i} \leq \sum_{i=1}^{|S|} \frac{w_S}{|S| - i + 1} = w_s \left(\frac{1}{|S|} + \frac{1}{|S| - 1} + \cdots + 1 \right) = w_S \cdot H_{|S|} \leq w_S \cdot H_\Delta$$

As y/H_Δ is dual feasible solution, then

Theorem 10.3

Greedy algorithm is an H_Δ-approximation algorithm.

10.2 LP-Rounding Algorithms for Set Cover

Observation 1 If $x^*_S \geq \frac{1}{c}$, and we set $\hat{x}_S = 1$ in an integer solution then cost increase by a factor of at most c (relative to cost of x^*)

Observation 2 For any element e, there exists some set S containing e s.t. $x^*_S \geq \frac{1}{B}$, because there are at most B terms on the LHS of (1).

These two observations suggest LP-rounding B-approximation algorithm: Pick all sets S with $x^*_S \geq \frac{1}{B}$. In observation 2, we get a set cover. In observation 1, we get a solution of cost at most $B \cdot \mathrm{OPT}(LP)$. Thus a B-approximation.

LP-rounding $O(\ln n)$-approximation for set cover

where n is number of elements. Important technique: **Randomized Rounding**.

Idea Interpret $x_S^* \in [0,1]$ as a probability, then x_S^* is the probability that set S is picked.

(A) Pick each set S independently with probability x_S^*.

Issues

1. We may not cover every element.
2. Can only say that expected cost of sets picked is bounded:

$$\text{Expected cost} = \sum_S w_S \cdot \Pr[S \text{ is picked}] = \sum_S w_S \cdot x_S^* = \text{OPT}(LP)$$

We will *only* deal with issue (1): Can we bound $\Pr[e \text{ is not bounded}]$ for some element e, and hence bound the probability "we do not have a set cover". For an element $f \in U$,

$$\begin{aligned}
\Pr[f \text{ is not covered}] &= \Pr[\text{no set } S \text{ containing } f \text{ is picked}] && \text{sets are picked independently} \\
&= \prod_{S: f \in S} (1 - x_S^*) && \text{as } 1 - x \le e^{-x} \\
&\le \prod_{S: f \in S} e^{-x_S^*} \\
&= e^{-\sum_{S: f \in S} x_S^*} && \textstyle\sum_{S: f \in S} x_S^* \ge 1 \\
&\le e^{-1}
\end{aligned}$$

Suppose we repeat (A) for $2 \ln n$ rounds, each round is independent of all other rounds. Now

$$\begin{aligned}
\Pr[f \text{ is not covered at the end of } 2\ln n \text{ rounds}] &= \prod_{i=1}^{2\ln n} \Pr[f \text{ is not covered in round } i] \\
&\le \prod_{i=1}^{2\ln n} e^{-1} \\
&= e^{-2\ln n} \\
&= \frac{1}{n^2}
\end{aligned}$$

Basic fact (union bound): $P(A \cup B) \le P(A) + P(B)$. thus

$$\begin{aligned}
\Pr[\text{we don't have a set cover after } 2\ln n \text{ rounds}] &= \Pr[\exists f \in U \text{ s.t. } f \text{ is not covered after } 2\ln n \text{ rounds}] \\
&\le \sum_{f \in U} \Pr[f \text{ is not covered after } 2\ln n \text{ rounds}] \\
&\le n \cdot \frac{1}{n^2} \\
&= \frac{1}{n}
\end{aligned}$$

Then the expected cost of sets picked:

$$E[\text{cost of sets picked after } 2\ln n \text{ rounds}] \le \sum_{i=1}^{n \ln n} E[\text{cost of sets picked in round } i] = 2\ln n \cdot \text{OPT}(LP)$$

So we have a randomized algorithm such that

1. Algorithm has a probability of error, i.e., we do not have a set cover.
2. Can only say that expected cost of solution is at most $2\ln n \cdot \text{OPT}(LP)$.

Now we want to make error probability equal to zero. If after $2 \ln n$ rounds, sound element is not covered, then simply pick the min-weight set S_f containing f, and add that to our solution, i.e., the sets picked. Now

$$\begin{aligned} E[\text{cost of our solution}] &\leq 2 \ln n \cdot \text{OPT}(LP) + \sum_{f \in U} \Pr[f \text{ is not covered after } 2 \ln n \text{ rounds}] \cdot w_{S_f} \\ &\leq 2 \ln n \cdot \text{OPT}(LP) + n \cdot \frac{1}{n^2} \cdot \text{OPT}(LP) \\ &\leq \left(2 \ln n + \frac{1}{n}\right) \text{OPT}(LP) \end{aligned}$$

as $w_{S_f} \leq \text{OPT}(LP)$. This is because any feasible LP solution satisfies $\sum_{S:f \in S} x_S \geq 1$; then

$$\text{OPT}(LP) = \sum_{S:f \in S} x_S w_S \geq \sum_{S:f \in S} (\min_{S:f \in S} w_S) x_S = w_{S_f} \sum_{S:f \in S} x_S \geq w_{S_f}$$

10.3 Uncapacitated Facility Location

We are given a complete bipartite graph $G = (V = F \cup C, E)$ (so there is an edge ij $\forall i \in F, j \in C$). F is facilities, C is clients.

- Every facility i has an opening cost $f_i \geq 0$.
- For each facility $i \in F$, client $j \in C$, we have a connection/assignment cost $c_{ij} \geq 0$, which is the cost of assigning client j to facility i.

The goal is to choose a set $F' \subseteq F$ facilities to open, and assign each client $j \in C$ to an open facility $i(j) \in F'$ so as to minimize $\sum_{i \in F'} f_i + \sum_{j \in C} c_{i(j)j}$ which is the sum of *facility opening cost + client assignment/connection cost*.

Note that any open facility can serve any number of clients, this is uncapacitated.

Recall this is NP-complete. We then have *LP-relaxation*. i indexes facilities in F, j indexes clients in C. Variables:

- y_i indicates if facility i is open $\forall i \in F$.
- x_{ij} indicates if client j is assigned to facility i $\forall i \in F, j \in C$.

$$\begin{aligned} \min \quad & \sum_{i \in F} f_i y_i + \sum_{j \in C} \sum_{i \in F} c_{ij} x_{ij} && \text{(UFL-P)} \\ \text{s.t.} \quad & \sum_i x_{ij} \geq 1 \quad \forall j \in C && (1) \\ & x_{ij} \leq y_i \quad \forall i \in F, j \in C && (2) \\ & x, y \geq 0 \end{aligned}$$

(1) means every j is assigned to a facility. (2) means j is assigned to an open facility.

$$\begin{aligned} \max \quad & \sum_{j \in C} \alpha_j && \text{(UFL-D)} \\ \text{s.t.} \quad & \alpha_j - \beta_{ij} \leq c_{ij} \quad \forall i, j \\ & \sum_{j \in C} \beta_{ij} \leq f_i \quad \forall i \\ & \alpha, \beta \geq 0 \end{aligned}$$

11

UFL

Recall i indexes facilities in F, j indexes clients in C. Variables:

- y_i indicates if facility i is open $\forall i \in F$.
- x_{ij} indicates if client j is assigned to facility i $\forall i \in F, j \in C$.

$$\begin{aligned}
\min \quad & \sum_{i \in F} f_i y_i + \sum_{j \in C} \sum_{i \in F} c_{ij} x_{ij} && \text{(UFL-P)} \\
\text{s.t.} \quad & \sum_i x_{ij} \geq 1 \qquad \forall j \in C && (1) \\
& x_{ij} \leq y_i \qquad \forall i \in F, j \in C && (2) \\
& x, y \geq 0
\end{aligned}$$

$$\begin{aligned}
\max \quad & \sum_{j \in C} \alpha_j && \text{(UFL-D)} \\
\text{s.t.} \quad & \alpha_j - \beta_{ij} \leq c_{ij} \quad \forall i, j \\
& \sum_{j \in C} \beta_{ij} \leq f_i \quad \forall i \\
& \alpha, \beta \geq 0
\end{aligned}$$

11.1 Economic Interpretation of Dual LP

α_j is amount j is willing to pay to get itself assigned to a facility.

First, suppose all facilities are free, i.e., have 0 cost, or facility is not $\underbrace{\text{charging any payment}}_{\beta_{ij}}$ from clients (to produce service). Then, each client j will want to get assigned to facility i' such that $c_{i'j} = \min_{i \in F} c_{ij}$, i.e., pays an amount $\min_{i \in F} c_{ij}$. So we get a lower bound of $\sum_{j \in C}(\min_{i \in F} c_{ij})$ on OPT.

To get a better lower bound, suppose each facility i charges an amount $\beta_{ij} \geq 0$ (share of its opening cost chared to j) from client j (to provide service). These cost shares have to satisfy the fairness condition $\sum_{j \in C} \beta_{ij} \leq f_i$ $\forall i \in F$.

Now, client j has to pay a *net cost* of $c_{ij} + \beta_{ij}$ to get assigned to facility i. Then client j would pay $\min_{i \in F}(c_{ij} + \beta_{ij})$ and so $LB(\beta) := \sum_{j \in C} \min_{i \in F}(c_{ij} + \beta_{ij})$ is a lower bound on OPT.

Any $\{\beta_{ij}\}_{i\in F, j\in C}$ cost-sharing scheme satisfying fairness condition gives $LB(\beta)$ as lower bound. So to get best lower bound, we want to

$$\begin{array}{ll} \max & \sum_{j\in C} \underbrace{\min_{i\in F}(c_{ij}+\beta_{ij})}_{:=\alpha_{ij}} \\ \text{s.t.} & \sum_{j\in C}\beta_{ij}\le f_i \quad \forall i\in F \\ & \beta\ge 0 \end{array} \quad \text{is equivalent to} \quad \begin{array}{lll} \max & \sum_{j\in C}\alpha_j & \\ \text{s.t.} & \sum_{j\in C}\beta_{ij}\le f_i & \forall i\in F \\ & \beta\ge 0 & \\ & \alpha_j\le \min_{i\in F}(c_{ij}+\beta_{ij}) & \forall j\in C \end{array}$$

The last constraint is equivalent to $\alpha_j \le c_{ij}+\beta_{ij}\ \forall j\in C, i\in F$. So the best LB is obtained by (UFL-D), where we add $\alpha\ge 0$ without any change in optimal value.

Complementary Slackness condition

1. (a) $x_{ij}>0 \Rightarrow \alpha_j = c_{ij}+\beta_{ij}$, so $c_{ij}\le\alpha_j$ (since $\beta_{ij}\ge 0$) $\forall i\in F, j\in C$.

   (b) $y_i>0\Rightarrow \sum_j \beta_j = f_i$

2. $\alpha_j>0\Rightarrow \sum_i x_{ij}=1$

   $\beta_{ij}>0\Rightarrow x_{ij}=y_i$

11.2 LP-rounding Algorithms for general assignment costs

Let (x^*,y^*) be the optimal solution to (UFL-P), (α^*,β^*) be the optimal solution to (UFL-D). Let $\text{OPT}(LP)$ be the common optimal value of (UFL-D), (UFL-P), so

$$\sum_i f_i y_i^* + \sum_{i,j} c_{ij}x_{ij}^* = \text{OPT}(LP) = \sum_j \alpha_j^*$$

Define $F_j^*=\{i\in F: x_{ij}^*>0\}$. (Note that (x^*,y^*) is potentially fractional solution, so $|F_j^*|$ will typically be more than 1)

We make several observations:

(X) $c_{ij}\le\alpha_j^*\ \forall i\in F_j^*$. This is the same CS condition 1(a)

(A) Suppose we have found a facility-set F' s.t. $F'\cap F_j^*\ne\varnothing\ \forall j\in C$. (*)

Then the cost of solution that opens F', and assigns each $j\in C$ to some facility in $F'\cap F_j^*$, is at most

$$\sum_{i\in F'} f_i + \text{total assignment cost} \le \sum_{i\in F'} f_i + \sum_{j\in C}\alpha_j^* = \sum_{i\in F'} f_i + \text{OPT}(LP)$$

(B) Finding a min-cost set F' satisfying (*) is a *set-cover problem* where each facility i is a set with weight f_i, that covers all clients $j\in C$ s.t. $i\in F_j^*$. So universe is C.

(C) There is a feasible solution y^* to LP-relaxation (SC-P) for the set-cover instance in (B) of cost at most $\text{OPT}(LP)$. This is because for any $j\in C$, $\sum_{i\in F_j^*} y_i^* \ge \sum_{i\in F_j^*} x_{ij}^* = \sum_{i\in F} x_{ij}^* \ge 1$, and $cost(y^*) = \sum_{i\in F} f_i y_i^* \le \text{OPT}(LP)$.

(D) If $\mathcal{A}$ is a ρ-approx. algorithm for set-cover that returns a solution of cost at most $\rho\cdot\text{OPT(SC-P)}$ (LP-relative approximation), then we can use this on set cover instance in observation (B) to get a facility-set F' s.t. $\sum_{i\in F'} f_i \le \rho\sum f_i y_i^* \le \rho\cdot\text{OPT}_{LP}$. And F' satisfies (*), then we get a UFL solution of total cost at most $(\rho+1)\,\text{OPT}_{LP}$. This gives us a $(\rho+1)$-approximation.

All algorithms we have seen for set cover are LP-relative approximation algorithm. In particular, using greedy algorithm for set cover, we obtain an (H_n+1)-approximation algorithm for UFL, where $H_n+1\approx O(\log n)$ and $n=|C|$.

With general c_{ij}'s ($c_{ij} \geq 0$), *any* LP-rounding approximation algorithm *must* assign each client j to some facility i with $x^*_{ij} > 0$ (i.e., we must open F' satisfying (*)). This is because if $x^*_{ij} = 0$, then we can increase c_{ij} arbitrarily without affecting LP solution, i.e., no handle on c_{ij} if $x^*_{ij} = 0$.

There are instances, where any algorithm satisfying (*) must have cost $\approx \ln n \cdot \text{OPT}(LP)$.

Upshot To do better than $\ln n$, algorithm must sometimes assign a client j to a facility i with $x^*_{ij} = 0$, and so we need some structure/assumption on the c_{ij}'s so that this is not incompatible with having a bounded approximation ratio.

11.3 Metric UFL

Recall define $F^*_j = \{i : x^*_{ij} > 0\}\ \forall j \in C$. Suppose we find F' s.t. $F' \cap F^*_j \neq \varnothing\ \forall j \in C$ $(*)$

To get an improved algorithm, we need to impose some structure on c_{ij}'s.

Metric UFL

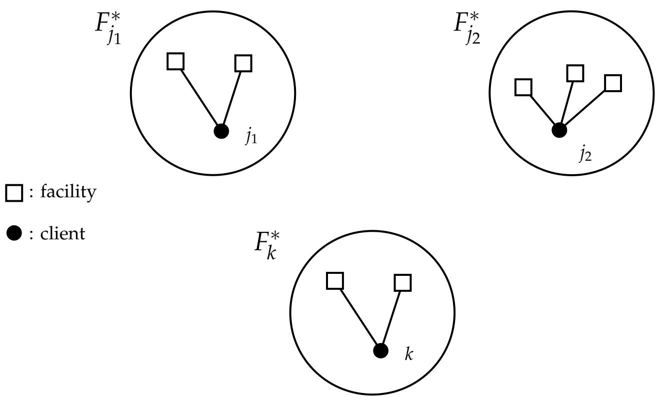
Connection costs satisfy the triangle-inequality, i.e., c_{ij}'s form a metric. That is,

$$\forall i, i' \in F, \forall j, j' \in C : \quad c_{i'j} \leq c_{ij} + c_{ij'} + c_{i'j'}$$

It's convenient to extend c_{ij}'s to distance/costs $c_{uv}\ \forall u, v \in F \cup C$ by *defining* c_{uv} to be the shortest-path distance between u and v in the graph $G = (F \cup C, E = \{ij, i \in F, j \in C\})$ with $\{c_{ij}\}_{i \in F, j \in C}$ edge cost.

Note that for $i \in F, j \in C$, the shortest i-j path is the single edge ij, due to the triangle-inequality.

Observation If all F^*_j-sets were pairwise disjoint (i.e., $F^*_j \cap F^*_k = \varnothing\ \forall j, k \in C, j \neq k$), then we can find F^* satisfying $(*)$ of cost $\leq \sum_i f_i y^*_i \leq \text{OPT}(LP)$. By opening cheapest (in terms of f_i) facility in each F^*_j set.



Goal Pick a subset $C' \subseteq C$ of clients such that

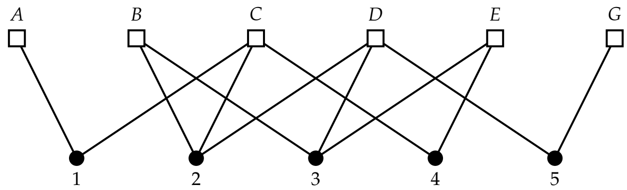
1. F^*_j sets for all $j \in C'$ are disjoint.
2. any $k \in C - C'$ is "close" to a client in C'

Algorithm 12: Clustering algorithm to find C'

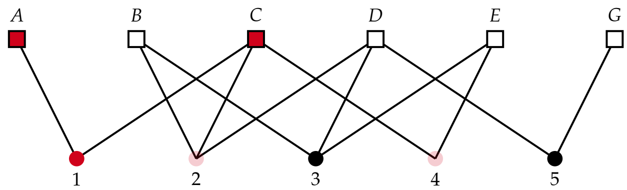
$L \leftarrow$ list of all clients sorted in increasing α_j^*-order (break ties arbitrarily)
$C' \leftarrow \varnothing$
while $L \neq \varnothing$ **do**
  Let j be first client in L. $C' \leftarrow C' \cup \{j\}$, $L \leftarrow L - \{j\}$.
  for $k \in L$ **do**
    if $F_k^* \cap F_j^* \neq \varnothing$ **then**
      $L \leftarrow L - \{k\}$, set $\text{nbr}(k) = j$
return C'

Example:

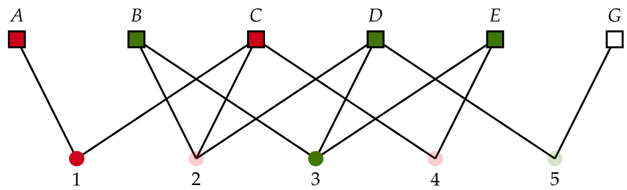
Suppose we have $\alpha_1^* \leq \alpha_2^* \leq \alpha_3^* \leq \alpha_4^* \leq \alpha_5^*$. There is an edge ij whenever $x_{ij}^* > 0$ (i.e., $i \in F_j^*$). As usual, we use circle to denote client, and square to denote facility.



First add $j = 1$, and $2, 4$ get removed.



Next add $j = 3$, and 5 gets removed.



So we have $C' = \{1, 3\}$ and $\text{nbr}(2) = 1$, $\text{nbr}(5) = 3$, $\text{nbr}(4) = 1$.

Algorithm 13: Metric UFL

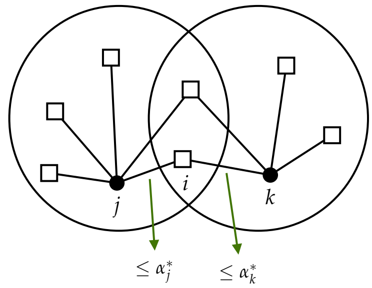
Run clustering algorithm to find C'
$F' \leftarrow \varnothing$
for $j \in C'$ **do**
  Open $i \in F_j^*$ with smallest f_i, i.e., add i to F'
  Assign j to i, and assign every $k \in C - C'$ with $\text{nbr}(k) = j$ to i
return F' *and corresponding client assignment*

Now let's analyze the approximation guarantee.

Claim 11.1

If $k \in C - C'$, and $\text{nbr}(k) = j \in C'$, then $c_{jk} \leq \alpha_j^* + \alpha_k^*$, and $\alpha_j^* \leq \alpha_k^*$.

Proof:



Since $\text{nbr}(k) = j$, we have $F_j^* \cap F_k^* \neq \emptyset$. Let $i \in F_j \cap F_k^*$, so by CS conditions, we have $c_{ij} \leq \alpha_j^*$ and $c_{ik} \leq \alpha_k^*$. And so $c_{jk} \leq c_{ij} + c_{ik} \leq \alpha_j^* + \alpha_k^*$. Since $\text{nbr}(k) = j$, j comes before k in L, thus we have $\alpha_j^* \leq \alpha_k^*$ □

Lemma 11.2

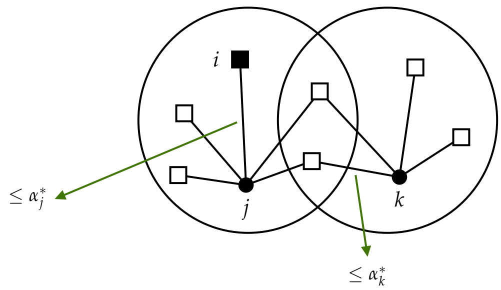
(a) If $j \in C'$, then it is assigned to a facility $i \in F'$ such that $c_{ij} \leq \alpha_j^*$.

(b) If $k \in C - C'$, then it is assigned to $i \in F'$ such that $c_{ij} \leq 3\alpha_k^*$.

Proof:

For part (a), it's easy to see that since $j \in C'$, it is assigned to some $i \in F_j^*$, thus $c_{ij} \leq \alpha_j^*$.

For part (b),



Let $j = \text{nbr}(k)$. So k is assigned to some $i \in F_j^*$. By Claim 11.1, $c_{jk} \leq \alpha_j^* + \alpha_k^*$ and $\alpha_j^* \leq \alpha_k^*$. We have

$$c_{ik} \leq c_{jk} + c_{ij} \leq (\alpha_j^* + \alpha_k^*) + \alpha_j^* \leq 2\alpha_j^* + \alpha_k^* \leq 3\alpha_k^*$$

□

Lemma 11.3

Cost of opening facilities is at most $\sum_i f_i y_i^*$.

Proof:

The cost of cheapest facility i in F_j^* is $f_i \leq \sum_{i' \in F_j^*} f_{i'} x_{i'j}^*$ since $f_i \leq f_{i'}$ $\forall i' \in F_j^*$ and $\sum_{i' \in F_j^*} x_{i'j}^* \geq 1$. So we have

$$f_i \leq \sum_{i' \in F_j^*} f_{i'} x_{i'j}^* \leq \sum_{i' \in F_j^*} f_{i'} y_{i'}^* \qquad (\spadesuit)$$

We then add up ($\spadesuit$) for all $j \in C'$ (ignoring gray part):

$$\sum_{i \in F'} f_i \leq \sum_{j \in C'} \sum_{i \in F_j^*} f_{i'} y_{i'}^* \leq \sum_{i \in F} f_i y_i^*$$

since F_j^* sets are disjoint $\forall j \in C'$. □

Theorem 11.4

Total cost is at most $4 \cdot \text{OPT}(LP)$.

which implies this is a 4-approximation algorithm.

Proof:
By Lemma 11.3, total facility-opening cost is at most $\sum f_i y_i^* \leq \text{OPT}(LP)$. By Lemma 11.2, total assignment cost is at most $3 \sum_j \alpha_j^* = 3 \cdot \text{OPT}(LP)$. So total cost is at most $4 \cdot \text{OPT}(LP)$. □

12

Computational Methods for (General) Integer Programs

12.1 Relaxation Methods

Consider an optimization problem

$$\max f(x) \qquad \text{s.t.} \quad x \in X \subseteq \mathbb{R}^n \tag{P}$$

relaxation

We say that

$$\max g(x) \qquad \text{s.t.} \quad x \in G \subseteq \mathbb{R}^n \tag{R}$$

is a relaxation of (P) if

1. $X \subseteq G$, and
2. $f(x) \leq g(x)\ \forall x \in X$.

Suppose (P) and (R) both have optimal solutions. Let $x^{(1)}$ be optimal solution to (P), $x^{(2)}$ be optimal solution to (R).

Observation 12.1

$\mathrm{OPT}(R) \geq \mathrm{OPT}(P)$.

Proof:

$$\mathrm{OPT}(P) = f(x^{(1)}) \overset{*}{\leq} g(x^{(1)}) \overset{\star}{\leq} g(x^{(2)}) = \mathrm{OPT}(R)$$

$(*)$: property 2

$(\star)$: $x^{(1)} \in X \subseteq G$, then by property (1), $x^{(2)}$ optimal solution to (R). □

Observation 12.2

Suppose $f = g$. Then if $x^{(2)} \in X$, we have $x^{(2)}$ optimal solution to (P).

Proof:
$f(x^{(2)}) = g(x^{(2)}) = \mathrm{OPT}(R) \geq \mathrm{OPT}(P)$ by Observation 12.1. And we know $x^{(2)}$ feasible to (P). Thus $x^{(2)}$ is optimal solution to (P). □

Let's examine some examples of relaxations.

Example: LP-relaxation

Suppose

$$\max c^T x \qquad \text{s.t.} \quad \underbrace{Ax \leq b,\ x \text{ integer}}_{X} \tag{P}$$

is an integer program. Then its LP-relaxation is

$$\max c^T x \qquad \text{s.t.} \quad \underbrace{Ax \leq b}_{G} \tag{R}$$

- $G \supseteq X$ since we have only *dropped* integrality constraints.
- $f(x) = g(x) = c^T x$

is indeed a relaxation under our definition.

Example: Lagrangian Relaxation

$$\begin{aligned} \max \quad & c^T x \\ \text{s.t.} \quad & Dx \leq d \\ & x \in X_1 \end{aligned} \tag{P}$$

where $Dx \leq d$ has m inequalities. So feasible region of (P) is

$$X = \{x : Dx \leq d, x \in X_1\}$$

Think of

- $Dx \leq d$ "complicating" constraints
- $x \in X_1$: "easy" constraints (e.g., $\max c^T x \quad \text{s.t. } x \in X_1$ is easier to solve)

Let $\lambda \in \mathbb{R}^m_+$, i.e., $\lambda_1, \lambda_2, \ldots, \lambda_m \geq 0$. Consider the problem $LR(\lambda)$:

$$\max c^T x + \lambda^T (d - Dx) \qquad \text{s.t.} \quad x \in X_1$$

where $\lambda^T(d - Dx)$ is multiplying i-th constraint of $d - Dx \geq 0$ by λ_i, and adding these terms to objective function. Also observe that $\lambda^T d$ is constant, then this is equivalent to

$$\lambda^T d + \max(c^T - \lambda^T D)x \qquad \text{s.t.} \quad x \in X_1 \tag{$LR(\lambda)$}$$

Lemma 12.3

$LR(\lambda)$ is a relaxation of (P), for any $\lambda \geq 0$.

Proof:
For property (1), feasible region of $LR(\lambda)$ is X_1, and by definition $X \subseteq X_1$.

For property (2), for any $x \in X$, we have $Dx \leq d$, i.e., $d - Dx \geq 0$. Since $\lambda \geq 0$, $\lambda^T(d - Dx) \geq 0$, and so objective function of $LR(\lambda)$ is $c^T x + \lambda^T(d - Dx) \geq c^T x$. □

$LR(\lambda)$ is called **Lagrangian Relaxation** of (P)

- with respect to $Dx \leq d$ constraints.

- obtained by dualizing $Dx \le d$ constraints.

So by Lemma 12.3, and Observation 12.1, $\text{OPT}(LR(\lambda)) \ge \text{OPT}(P)\ \forall \lambda \ge 0$. So

$$z_{LD} := \min_{\lambda \in \mathbb{R}^m_+} \text{OPT}(LR(\lambda)) = \min_{\lambda \in \mathbb{R}^m_+} \begin{pmatrix} \max & c^T x + \lambda^T (d - Dx) \\ \text{s.t.} & x \in X_1 \end{pmatrix}$$

is an upper bound on $\text{OPT}(P)$. This is bust upper bound obtained using $LR(\lambda)$. z_{LD} is called **Lagrangian dual** with respect to $Dx \le d$ constraints.

Example: Bounded-Degree Max-weight Spanning Tree Problem

Consider following Bounded-Degree Max-weight Spanning Tree Problem. Given connected undirected graph $G = (V, E)$, $\{w_e\}_{e \in E}$ edge weights. and degree bounds $\{b_v \ge 0, \text{ integer}\}_{v \in V}$. The goal is to find a spanning tree R of max weight such that every node v has degree at most b_v in T, i.e.,

$$\begin{array}{ll} \max & w(T) \\ \text{s.t.} & T \text{ is a spanning tree} \\ & |\delta_T(v)| \le b_v \quad \forall v \in V \end{array}$$

where $\delta_T(v) = \delta(v) \cap T$.

We introduce a notation. For a set $F \subseteq E$, the incidence vector of F, x^F, is the $\{0, 1\}$ vector in $\mathbb{R}^E$ given by

$$x^F_e = \begin{cases} 1 & \text{if } e \in F \\ 0 & \text{otherwise} \end{cases}$$

Ler $X_1 = \{x^T : T \text{ is a spanning tree of } G\}$. Our problem is

$$\begin{array}{ll} \max & w^T x \\ \text{s.t.} & x \in X_1 \\ & \sum_{e \in \delta(v)} x_e \le b_v \quad \forall v \in V \end{array}$$

where the second constraint is complicated DEGREE constraint. Then the Lagrangian Relaxation $LR(\lambda)$ obtained by dualizing DEGREE constraints.

$$\begin{aligned} &\max w^T x + \sum_{v \in V} \lambda_v (b_v - \sum_{e \in \delta(v)} x_e) && \text{s.t. } x \in X_1 \\ \equiv &\sum_{v \in V} \lambda_v b_v + \max \sum_{e = uv \in E} x_e (w_e - \lambda_u - \lambda_v) && \text{s.t. } x \in X_1 \end{aligned} \qquad (LR(\lambda))$$

Consider an integer program (P): $\max c^T x \quad \text{s.t. } Ax \le b$, x integer.

Is there an "ideal relaxation" of (P), i.e., a relaxation that we know how to solve, and whose optimal value is $\text{OPT}(P)$? YES! We can solve (P) by solving a certain LP. BUT, this LP may have a huge number of constraints.

Recall, a set $Z \subseteq \mathbb{R}^n$ is convex if $\forall x, y \in Z$, $\forall \lambda \in [0, 1]$, the point $\lambda x + (1 - \lambda) y \in Z$. Empty set is convex set, and intersection of arbitrary family of convex sets is convex.

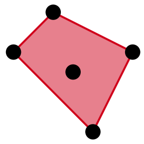
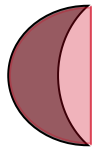
convex hull

Given a set $Z \subseteq \mathbb{R}^n$, the convex hull of Z, denoted $\text{conv}(Z)$ is the smallest convex set containing Z. I.e., if C is convex, $C \supseteq Z$, then $C \supseteq \text{conv}(Z)$.

Not hard to show that

$$\text{conv}(Z) = \bigcap_{C \supseteq Z : C \text{ convex}} C$$

Example:

(a)  (b) 

(a) $Z = \{\text{all black dots}\}$, and $\text{conv}(Z)$ is red region.

(b) Z is black region, and $\text{conv}(Z)$ is red region.

(c) $Z = \text{conv}(Z)$ if and only if Z is convex.

Theorem 12.4

Let (P) be the integer program: $\max c^T x \quad \text{s.t.}\, Ax \leq b,\ x$ integer, where A, b rational (i.e., all entries are rational numbers). Let X be feasible region of (P). Then

(a) $\text{conv}(X)$ is a polyhedron. (i.e., a set of the form $\{x : Mx \leq b\}$, or feasible region of some LP)

(b) Consider (R): $\max c^T x \quad \text{s.t.}\, x \in \text{conv}(X)$. Then

- (R) is a relaxation of (P)
- (R) is an LP, and $\text{OPT}(R) = \text{OPT}(P)$

So (R) is "ideal relaxation" of (P).

Proof:
It's bit hard to prove without introducing more convexity theories. □

Catch Describing $\text{conv}(X)$ may require a huge number of inequalities.

Lemma 12.5

Let $X \subseteq \mathbb{R}^n$, with $|X|$ finite. Then $\text{conv}(X)$ is polyhedron.

$\max w^T x$ s.t. $x \in \text{conv}(Z) = \max w^T x$ s.t. $x \in Z$.

Proof:
Skipped, partially because we run out of time, and no need to prove it. □

Lemma 12.6

Consider (P): $\max c^T x$ s.t. $x \in Z$, and its relaxation (R): $\max c^T x$ s.t. $x \in \text{conv}(Z)$. Then if x^* is optimal solution to (P), we also have x^* optimal solution to (R).

Proof:
Consider $C = \{x \in \mathbb{R}^n : c^T x \leq c^T x^*\}$ which is a halfspace. Then C is convex, $C \supseteq Z$, by definition of x^*. So $C \supseteq \text{conv}(Z)$, i.e., $c^T x \leq c^T x^*\ \forall x \in \text{conv}(Z)$. Also, $x^* \in Z \subseteq \text{conv}(Z)$. So x^* is optimal solution to (R). □

Theorem 12.7

Suppose (P): $\max c^T x$ s.t. $Dx \le d, x \in X_1$. Suppose $\text{conv}(X_1)$ is a polyhedron (e.g., if X_1 is finite or $X_1 = \{x : Ax \le b, x \text{ integer, where } A, b \text{ are rational}\}$). Then

$$z_{LD} = \max c^T x \quad \text{s.t. } Dx \le d, x \in \text{conv}(X_1)$$

12.2 Branch and Bound (BnB) Method

It is an enumerative/recursive method for solving based on a divide-and-conquer approach that exploits relaxations.

Goal Solve the optimization problem (P): $\max f(x)$ s.t. $x \in X$.

$$\mathsf{BnB}((Q), LB)$$

where

- (Q) is current subproblem that we need to solve: $\max f(x)$ s.t. $x \in X_Q$ where $X_Q \subseteq X$.
- LB is the lower bound on $\text{OPT}(P)$; whenever we find a feasible solution $\hat{x}$ to (P), we update $LB \leftarrow \max(LB, f(\hat{x}))$.

Initial call of BnB: $\mathsf{BnB}((P), LB := -\infty)$ as we haven't found any feasible solution. The steps are as follows

Algorithm 14: Branch and Bound: $\mathsf{BnB}((Q), LB)$

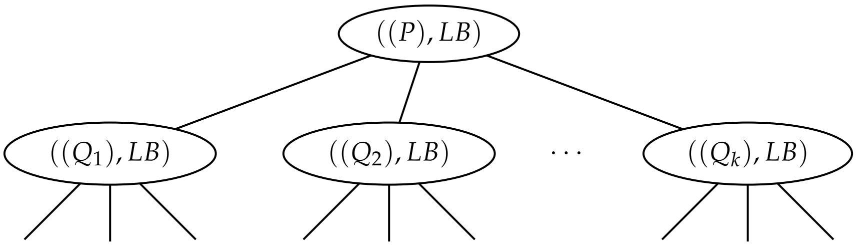
1 Solve a relaxation (R): $\max g(x)$ s.t. $x \in G$ of (Q). `// Assume (R) is not unbounded.`
if *(R) is infeasible* **then** STOP, return (Q) is infeasible, i.e., $X_Q = \varnothing$.
else $z_R := \text{OPT}(R)$, $x^{(R)}$ be the optimal solution to (R).

2 **if** $z_R \le LB$ **then**
STOP. `// Cannot get a better solution in` X_Q `than what we already have. This is called` **`Pruning`**

3 **else**
Use $x^{(R)}$ to find a optimal solution to (Q).
- If $f = g$, and $x^{(R)} \in X_Q$ (feasible region of (Q)), then by Observation 12.2, $x^{(R)}$ is optimal solution to (Q).
- If we can't deduce from $x^{(R)}$ that we have found an optimal solution to (Q), then **Branch**: Partition X_Q into $X_1 \cup \cdots \cup X_k$ (branching strategy) and recursively call BnB to find best solution in each X_i. I.e., call $\mathsf{BnB}((Q_i) : \max f(x) \text{ s.t. } x \in X_i, LB)$. `// Note: LP is updated whenever we found a feasible solution to (P), and so we always maintains the best solution found so far. So at the end` $LB = \text{OPT}(P)$

4 **return** *LB (and solution whose objective value is LB)*

The execution of BnB algorithm is often depicted by a **BnB tree**. A convenient way of descending execution of BnB method:



Issues:

1. We may create a lot of subproblems, e.g., an exponential number of subproblems.
2. Do we terminate? After how many iterations? Depends strongly on quality of bounds (i.e., z_R bounds), i.e., on quality of relaxations.

BnB using LP-relaxations Say (P) is the integer program, and also (Q) integer program. Here, relaxation (R) be LP-relaxation of (Q).

So if $x^{(R)}$, an optimal solution to (R), is integral (i.e., in X_Q), then $x^{(R)}$ is optimal solution to (Q).

Otherwise, a common and natural branching strategy is to pick one (or more) fractional variable(s) and branch based on the value of that variable(s) and branch based on the value of that variable(s) in $x^{(R)}$. For example, $x_j^{(R)} = 2.8$ which is non-integer, then we create the subproblems (**variable branching**):

(Q_1): (Q) + "$x_j \leq 2$"

(Q_2): (Q) + "$x_j \geq 3$"

Clearly we have $X_{Q_1} \cup X_{Q_2} = X_Q$ and $X_{Q_1} \cap X_{Q_2} = \varnothing$.

Knapsack Problem

Given n items; each item i has a value $v_i \geq 0$, integer weight $a_i \geq 0$. There is a knapsack of integer capacity $B \geq 0$. Goal is to find a max-value set of items that fit in the knapsack, i.e., whose total weight is at most B.

It can be formulated as following IP:

$$\begin{array}{rll} \max & \sum_{i=1}^{n} v_i x_i & \\ \text{s.t.} & \sum_{i=1}^{n} a_i x_i \leq B & \\ & x_i \in \{0,1\} & \forall i = 1,\dots,n \end{array} \tag{K-IP}$$

And its LP relaxation

$$\begin{array}{rll} \max & \sum_{i=1}^{n} v_i x_i & \\ \text{s.t.} & \sum_{i=1}^{n} a_i x_i \leq B & \\ & 0 \leq x_i \leq 1 & \forall i = 1,\dots,n \end{array} \tag{K-LP}$$

We then can solve knapsack via BnB using LP-relaxations and variable branching. We make several remarks and observations.

First observation is the following theorem and corollary:

Theorem 12.8

The following algorithm finds an optimal solution to (K-LP).

Sort items in decreasing order of $\rho_i = v_i/a_i$, and pack items in this order, subject to feasibility. I.e., let ℓ be the largest index (under sorted order) s.t. $\sum_{i=1}^{\ell} a_i \le B$. Set $x_1 = x_2 = \cdots = x_\ell = 1$, and

$$x_{\ell+1} = \text{fraction that fills up knapsack} = \frac{B - (a_1 + \cdots + a_\ell)}{a_{\ell+1}} < 1,$$

and $x_{\ell+2} = \cdots = x_n = 0$.

Corollary 12.9

There is an optimal solution to (K-LP) with at most one fractional variable (say $x_{\ell+1}$). This gives us a unique choice for branching variable, and the two branches will be

$$\begin{cases} x_{\ell+1} = 1 & (\text{i.e., } x_{\ell+1} \ge 1) \\ x_{\ell+1} = 0 & (\text{i.e., } x_{\ell+1} \le 0) \end{cases}$$

Observation 2 In a subproblem, some variables are fixed to be 0 or 1. Thus this is just another knapsack instance with

- some residual knapsack capacity B' (i.e., capacity left after we have included all items i with $x_i = 1$)
- some residual set of items S (i.e., items whose variables are not yet fixed to 0 or 1)

Will list subproblem by specifying B' and S.

Observation 3 Ordering of items in set S for a subproblem easily obtained from the ordering for all items, which needs to be computed only once.

Will specify optimal solution to LP-relaxation by indicating the fractional variable, and the variables whose values are 1.

Observation 4 From the optimal solution given by Theorem 12.8, we obtain a feasible (integer) solution

$$x_1 = \cdots = x_\ell = 1, \quad x_{\ell+1} = \cdots = x_n = 0$$

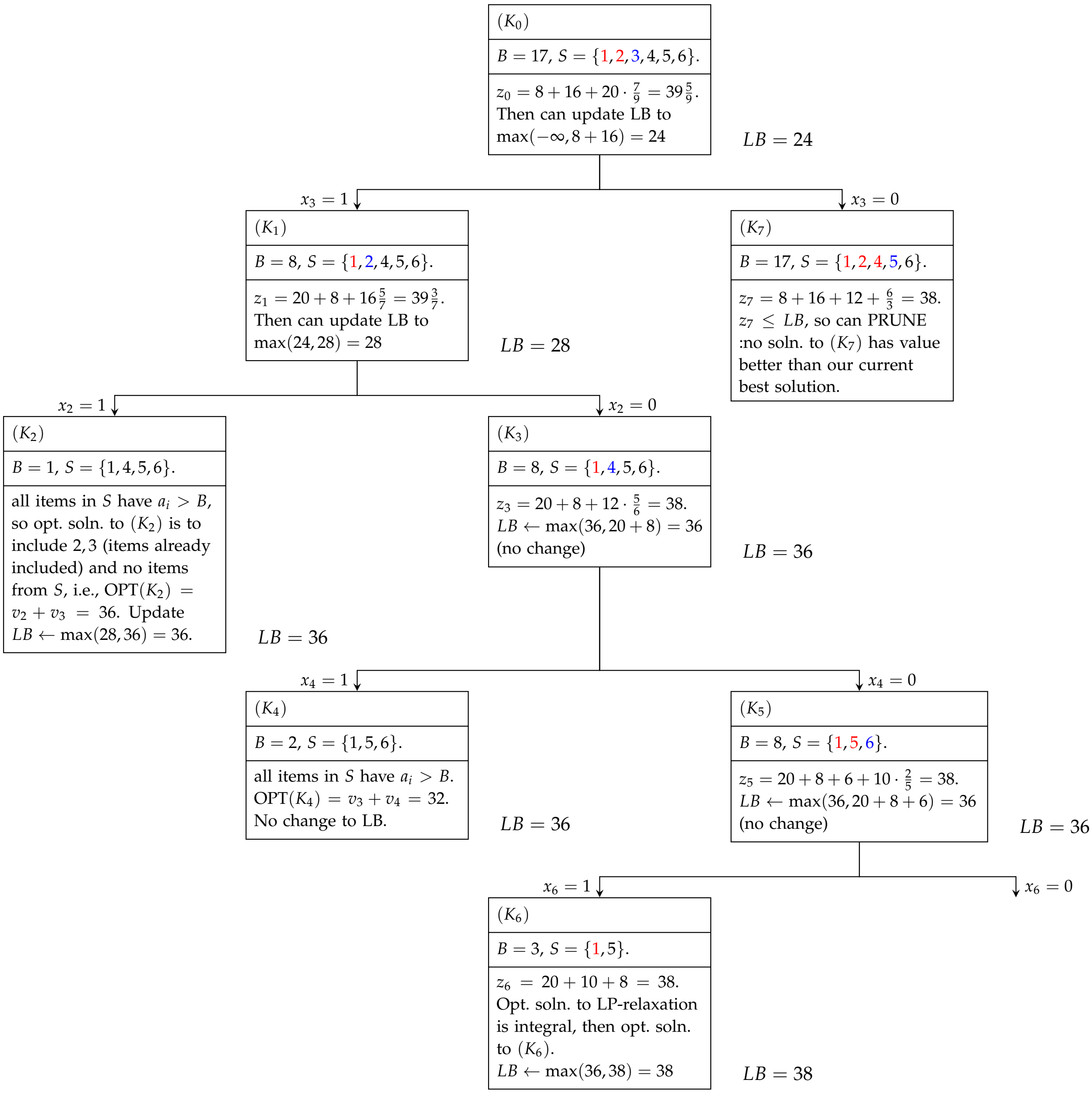
to knapsack, which can be used to update LP.

Now let's go through a concrete example. The knapsack capacity is 17. The items are as follows with v_i and a_i:

i	1	2	3	4	5	6
v_i	8	16	20	12	6	10
a_i	3	7	9	6	3	5
$\rho_i = v_i/a_i$	$8/3$	$16/7$	$20/9$	2	2	2

Let (K_i) denote residual knapsack subproblem, and z_i be the optimal value of LP-relaxation of (K_i). Also note that we use color red to denote $x_i = 1$, and color blue to denote it is a fractional variable.

Initially $LB = -\infty$



LB matches z_5, thus solution giving LB is also optimal solution to (K_5).

LB matches z_3, thus have found an optimal solution to (K_3).

Also means that we have found optimal solution to (K_1), since we have the optimal solutions to (K_2), (K_3):

$$\mathrm{OPT}(K_1) = \max(\mathrm{OPT}(K_2), \mathrm{OPT}(K_3)) = \max(36, 38) = 38.$$

So only need to explore $x_3 = 0$ branch of root node as shown above.

Then BnB terminates with optimal value $= LB = 38$, optimal solution (solution yielding value $= LB$):

$$x_3^* = x_6^* = 1, \quad x_1^* = 1, \quad x_2^* = x_4^* = x_5^* = 0$$

13

BnB using Lagrangian Relaxation

Recall

Theorem 12.7

If $\text{conv}(X_1)$ is a polyhedron, then

$$z_{LD} = \underbrace{\max c^T x \quad \text{s.t. } Dx \leq d, x \in \text{conv}(X_1)}_{(*)}$$

We can use z_{LD} to obtain an upper bound on (P) in BnB, or $z(\lambda) := \text{OPT}(LR(\lambda))$. Working with z_{LD} directly turns out to be expensive, but what works out better is using $z(\lambda)$ for suitable λ's. Before discussing this, we discuss some approaches for computing z_{LD}, if we chose to work this this upper bound.

13.1 Computing z_{LD}

- Sometimes $(*)$ may be equivalent to LP-Relaxation. Can sometimes be inferred using Theorem 12.7. And in this case, the λ's yielding optimal solution to (LD) can be obtained by solving dual of LP-relaxation of (P).
- **Column generation**. If $|X_1|$ is finite, say $X_1 = \{x^{(1)}, \ldots, x^{(k)}\}$, then we can show

  $$\text{conv}(X_1) = \left\{ \sum_{i=1}^{k} \lambda_i x^{(i)} : \lambda \geq 0, \sum_{i=1}^{k} \lambda_i = 1 \right\}$$

  So we can encode $(*)$ as

  $$\begin{aligned} \max \quad & c^T(\textstyle\sum_{i=1}^k \lambda_i x^{(i)}) \\ \text{s.t.} \quad & D(\textstyle\sum_{i=1}^k \lambda_i x^{(i)}) \leq d \\ & \lambda \geq 0 \\ & \textstyle\sum_i \lambda_i = 1 \end{aligned}$$

  which is an LP with λ_i's as variables. But $|X_1|$ could be quite large, and so we may get an LP with a huge number of variables. So instead of working with the entire-set of variables up-front, a more effective approach is to add variables on the fly, as and when required.
- **Subgradient optimization**. We can use "subgradient" descent to optimize $z(\lambda)$, which we can show piecewise linear function. Start with some $\lambda = \lambda^{(0)}$, move in direction of "gradient" (or

more precisely, "subgradient", because $z(\lambda)$ is not differentiable). The idea is to start with some λ, and more in a suitable direction (e.g., negative gradient) to decrease $z(\cdot)$.

In practice, do not compute z_{LD} as it's time consuming, but run some iterations of a subgradient-style method to get "good" λ's and use $z(\lambda)$ as upper bound.

Compared to working with the upper bound, it is now easy to compute upper bounds: for any $\lambda \in \mathbb{R}^m_+$, $z(\lambda)$ yields an upper bound. So as noted above, seek to compute some "good" λ's by running, say, a few iterations of subgradient descent.

Also, λ's computed for last subproblem, can be used for the next subproblem. So *upshot*: can get upper bounds very quickly for our subproblem. But Branching Strategy is not straightforward. Unlike when using LP relaxations, we may now obtain an infeasible, but integral vector (e.g., spanning tree violating degree bounds), so how do we branch?

13.2 Example: Traveling Salesman Problem

Given an undirected graph $G = (V, E)$, nonnegative edge costs $\{c_e\}_{e\in E}$, find a min-cost edge *cycle that visits every node exactly once*, which is called a TSP tour.

$x_e = 1$ if e is part of TSP $\forall e \in E$. $\{1\}$ is some fixed node set, or tour starting node.

$$\min \sum_{e\in E} c_e x_e \tag{TSP}$$

$$\text{s.t.} \sum_{e\in\delta(v)} x_e = 2 \qquad \forall v \in V - \{1\} \tag{1}$$

$$\sum_{e\in\delta(1)} x_e = 2 \tag{2}$$

$$\sum_{e\in E(S)} x_e \le |S| - 1 \qquad \forall \varnothing \subsetneq S \subsetneq V \tag{3}$$

$$\sum_{e\in E} x_e = n = |V| \tag{4}$$

$$x_e \in \{0, 1\} \qquad \forall e \in E$$

Here constraints (1) and (2) encode the degree constraints. Let F be a TSP tour. If we consider any nonempty strict subset S of V, $F \cap E(S)$ must be acyclic, and this is encoded by constraint (3).

Let X_1 denote the constraints other than (1). (TSP) is a minimization problem, so the Lagrangian relaxation will provide a *lower bound* on OPT(T). Then Lagrangian Relaxation with respect to (1) is

$$\min \sum_e c_e x_e + \sum_{v\neq 1} \lambda_v \left(2 - \sum_{e\in\delta(v)} x_e\right) \qquad \text{s.t. } x \in X_1$$

$$\min \quad 2\sum_{v\neq 1} \lambda_v + \sum_{e=uv\in E} (c_e - \lambda_u - \lambda_v) x_e \qquad \text{s.t. } x \in X_1 \tag{$LR(\lambda)$}$$

Here X_1 is the set of **1-trees**: consist of spanning tree on $V - \{1\}$, exactly 2 edges incident to 1.

For an edge $e = (u, v)$, we define $c_e(\lambda) := c_e - \lambda_u - \lambda_v$, which is the cost of e in the objective function of $(LR(\lambda))$ under the given λ-vector. We denote $z(\lambda) :=$ OPT$(LR(\lambda))$, and this is a lower bound on OPT(TSP). Hence we have the following.

Theorem 13.1

Suppose F is a TSP tour such that $c(F) = z(\lambda)$ for some λ. Then F is an optimal TSP tour.

The corresponding Lagrangian dual problem seeks to find the best lower bound, and is given by

$$\max_{\lambda=(\lambda_v)_{v\neq 1}} z(\lambda) \quad \equiv \quad \max_{\lambda=(\lambda_v)_{v\neq 1}} \left(\min \quad 2\sum_{v\neq 1} \lambda_v + \sum_{e=(u,v)} (c_e - \lambda_u - \lambda_v)x_e \quad \text{s.t.} \quad x \in X_1 \right) \quad \text{(LD)}$$

In general, at a branch-and-bound node, we will have fixed some x_e variables to 0 and some to 1, and the subproblem that we wish to solve is to find the minimum c-cost TSP solution that is consistent with the fixed variables. In the corresponding Lagrangian relaxation, we seek to find the minimum $c(\lambda)$-cost 1-tree that includes the edges whose x_e's are fixed to 1, and does not include the edges whose xes are fixed to 0. To find such a 1-tree, we use the following result.

Theorem 13.2

Given $G = (V, E)$, edge costs, disjoint sets F, Z, where F is the set of edges whose x_es are fixed to 1, Z is set of edges whose x_es are fixed to 0. Can compute a min-cost 1-tree T such that $T \supseteq F$, $T \cap Z = \varnothing$ (OR detect that no such 1-tree exists).

Theorem 13.2 (detailed)

Let $G = (V, E)$ be a connected undirected graph with $|\delta(1)| \geq 2$. Let F, Z be disjoint subsets of E, where F is the set of edges whose x_es are fixed to 1, Z is set of edges whose x_es are fixed to 0. Denote $G' = (V \setminus \{1\}, E(V \setminus \{1\}) \setminus Z)$.

(i) If $|\delta(1) \cap F| > 2$ or $|\delta(1) \setminus Z| < 2$ or G' is not connected, then there is no 1-tree T such that $T \supseteq F$, $T \cap Z = \varnothing$.

(ii) Otherwise, a minimum-cost 1-tree T such that $T \supseteq F$, $T \cap Z = \varnothing$ is obtained by taking a minimum-cost spanning tree of G' containing $F \setminus \delta(1)$, and adding to it the edges in $F \cap \delta(1)$ and the $2 - |F \cap \delta(1)|$ cheapest edges from $\delta(1) \setminus (F \cup Z)$.

BnB using LR for TSP

1. At a BnB node:
   - compute optimal 1-tree T given current λ_v's using Theorem 13.2
   - perform at most two "tweakings" of λ_v's:
     - If there exists v such that $|T \cap \delta(v)| > 2$, decrease λ_v by 1. By doing so, we *increase* the $c_e(\lambda)$-costs of all edges incident to v, which intuitively aims to decrease the degree of node v in the new optimal 1-tree.
     - If there exists v such that $|T \cap \delta(v)| < 2$, increase λ_v by 1.
     - If $|T \cap \delta(v)| = 2$, then we do not change λ_v.
2. At root node: start with $\lambda_v = 0 \; \forall v \in V - \{1\}$.
3. For next subproblem, will start off λ_v's obtained by tweaking the last λ_v's used for previous subproblem.
4. Branching Strategy: Suppose 1-tree T is *not* a TSP tour. Then it contains a cycle $C = \{e_1, \ldots, e_k\}$ that does not visit all nodes.

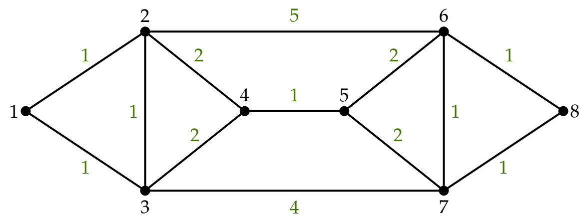
So a TSP tour must exclude at least one of $e_1, \dots, e_k$. So BRANCH into subproblems:

$$\begin{array}{ccccc}
x_{e_1} = 0 & x_{e_1} = 1 & x_{e_1} = 1 & \cdots & x_{e_1} = 1 \\
\vdots & x_{e_2} = 0 & x_{e_2} = 1 & \cdots & \vdots \\
 & \vdots & x_{e_3} = 0 & \cdots & \vdots \\
 & & \vdots & \ddots & x_{e_{k-1}} = 1 \\
 & & & & x_{e_k} = 0
\end{array}$$

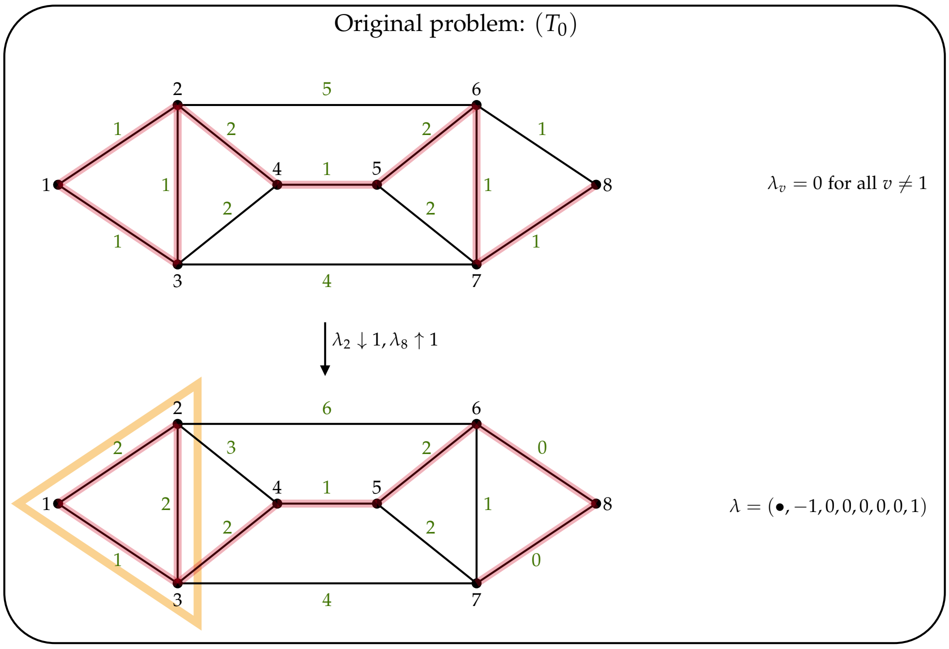
If some of the variables have been fixed previously, then we omit the subproblems where the above fixing of variables conflicts with the earlier fixing.

When to stop If we have a TSP tour T, and some λ_v's such that $c(T) = z(\lambda)$. Then T is optimal TSP tour.

Now let's examine a concrete example:

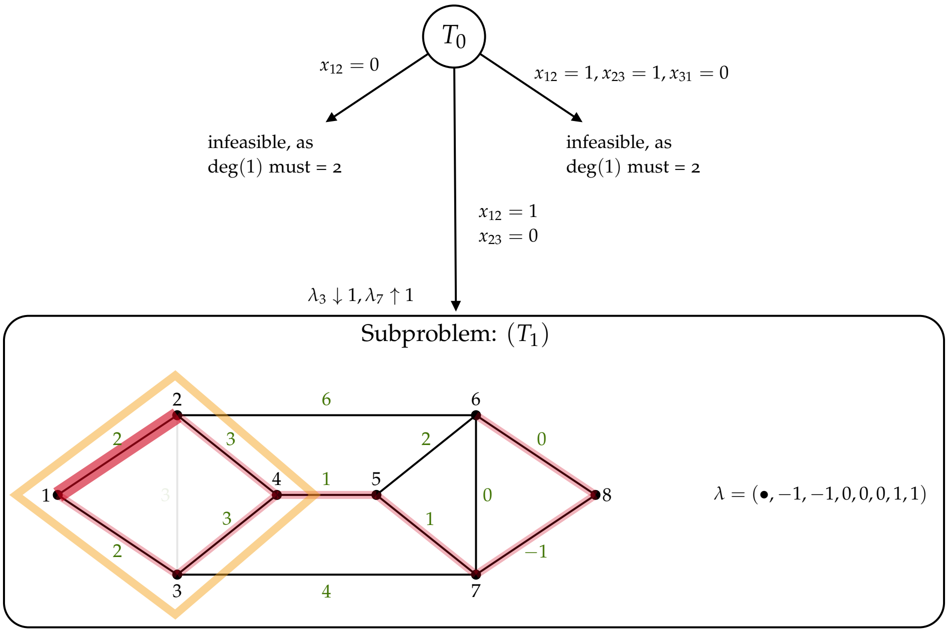


We start with root node T_0.

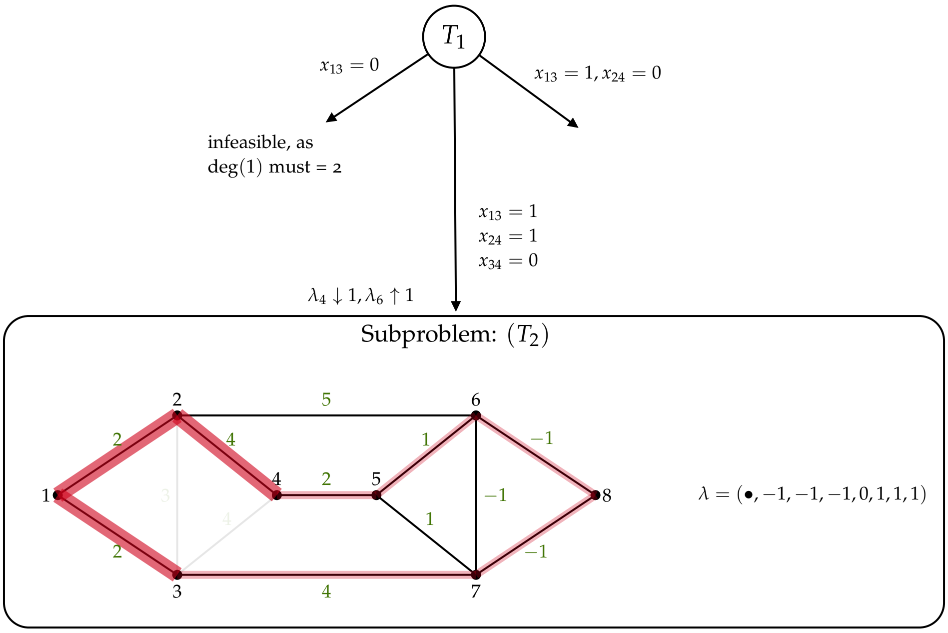


Root node

Then we branch based on cycle:



We indicate $x_e = 1$ by making edge thick, and indicate $x_e = 0$ by paling the edge. Then we branch based on cycle.



c-cost of tour found is 13. Lower bound on OPT(T_1) obtained by the optimal value of Lagrangian relaxation of (T_1):

$$\min \quad 2\sum_{v \neq 1} \lambda_v + \sum_{e=(u,v)} (c_e - \lambda_u - \lambda_v) x_e \quad \text{s.t.} \quad x \in X_1, \quad x_{12} = 1, x_{23} = 0 \qquad (LR^{(T_1)}(\lambda))$$

is 13. So this tour is the optimal tour for (T_1), and hence for (T_0).

Index