4. **Greedy Algorithms II**

- Dijkstra’s algorithm
- Minimum spanning trees
- Prim, Kruskal, Boruvka
- Single-link clustering
- Min-cost arborescences

---

**Single-pair shortest path problem**

**Problem.** Given a digraph $G = (V, E)$, edge lengths $\ell_e \geq 0$, source $s \in V$, and destination $t \in V$, find a shortest directed path from $s$ to $t$.

![Diagram of a digraph with shortest path highlighted](image1)

**Example.**

![Diagram of a digraph with shortest path highlighted](image2)

- **Source** $s$ and **destination** $t$.
- **Path length:** $9 + 4 + 1 + 11 = 25$

---

**Single-source shortest paths problem**

**Problem.** Given a digraph $G = (V, E)$, edge lengths $\ell_e \geq 0$, source $s \in V$, find a shortest directed path from $s$ to every node.

![Diagram of a digraph with shortest-paths tree](image3)

**Shortest-paths tree**
Q. Which kind of shortest path problem?
A. Single-destination shortest paths problem.

**Dijkstra’s algorithm (for single-source shortest paths problem)**

**Greedy approach.** Maintain a set of explored nodes $S$ for which algorithm has determined $d[u] =$ length of a shortest $s \rightarrow u$ path.

- Initialize $S \leftarrow \{ s \}$, $d[s] \leftarrow 0$.
- Repeatedly choose unexplored node $v \notin S$ which minimizes
  $$\pi(v) = \min_{e = (u,v) : u \in S} (d[u] + \ell_e)$$

  the length of a shortest path from $s$ to some node $u$ in explored part $S$, followed by a single edge $e = (u,v)$

  add $v$ to $S$, and set $d[v] \leftarrow \pi(v)$.
- To recover path, set $\text{pred}[v] \leftarrow e$ that achieves min.

**Shortest path applications**

- PERT/CPM.
- Map routing.
- Seam carving.
- Robot navigation.
- Texture mapping.
- Typesetting in LaTeX.
- Urban traffic planning.
- Telemarketer operator scheduling.
- Routing of telecommunications messages.
- Network routing protocols (OSPF, BGP, RIP).
- Optimal truck routing through given traffic congestion pattern.

Dijkstra’s algorithm: proof of correctness

**Invariant.** For each node \( u \in S \): \( d[u] = \) length of a shortest \( s \rightarrow u \) path.

**Prf.** [ by induction on \( |S| \) ]

**Base case:** \( |S| = 1 \) is easy since \( S = \{ s \} \) and \( d[s] = 0 \).

**Inductive hypothesis:** Assume true for \( |S| \geq 1 \).

- Let \( v \) be next node added to \( S \), and let \( (u, v) \) be the final edge.
- A shortest \( s \rightarrow u \) path plus \( (u, v) \) is an \( s \rightarrow v \) path of length \( \pi(v) \).
- Consider any other \( s \rightarrow v \) path \( P \). We show that it is no shorter than \( \pi(v) \).
- Let \( e = (x, y) \) be the first edge in \( P \) that leaves \( S \), and let \( P' \) be the subpath to \( x \).
- The length of \( P \) is already \( \geq \pi(v) \) as soon as it reaches \( y \):

\[
\ell(P) \geq \ell(P') + \ell_e \geq d[y] + \ell_e \geq \pi(y) \geq \pi(v) .
\]

**Dijkstra’s algorithm: efficient implementation**

**Critical optimization 1.** For each unexplored node \( v \notin S \), explicitly maintain \( \pi[v] \) instead of computing directly from definition

\[
\pi(v) = \min_{e = (u, v) : u \in S} \ d[u] + \ell_e
\]

- For each \( v \notin S \), \( \pi(v) \) can only decrease (because \( S \) only increases).
- More specifically, suppose \( u \) is added to \( S \) and there is an edge \( e = (u, v) \) leaving \( u \). Then, it suffices to update:

\[
\pi[v] \leftarrow \min \{ \pi[v], \pi[u] + \ell_e \}
\]

**Critical optimization 2.** Use a min-oriented priority queue \( (PQ) \) to choose an unexplored node that minimizes \( \pi[v] \).

**Dijkstra’s algorithm: which priority queue?**

**Performance.** Depends on \( PQ \): 
- Array implementation optimal for dense graphs. \( \Theta(n^2) \) edges
- Binary heap much faster for sparse graphs. \( \Theta(n \log n) \) edges
- 4-way heap worth the trouble in performance-critical situations.
- Fibonacci heap best in theory, but probably not worth implementing.

<table>
<thead>
<tr>
<th>priority queue</th>
<th>INSERT</th>
<th>DELETE-MIN</th>
<th>DECREASE-KEY</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered array</td>
<td>( O(1) )</td>
<td>( O(n) )</td>
<td>( O(1) )</td>
<td>( O(n^2) )</td>
</tr>
<tr>
<td>binary heap</td>
<td>( O(\log n) )</td>
<td>( O(\log n) )</td>
<td>( O(\log n) )</td>
<td>( O(m \log n) )</td>
</tr>
<tr>
<td>d-way heap (Johnson 1975)</td>
<td>( O(d \log_d n) )</td>
<td>( O(d \log_d n) )</td>
<td>( O(\log_d n) )</td>
<td>( O(m \log_d n) )</td>
</tr>
</tbody>
</table>
| Fibonacci heap (Fredman–Tarjan 1984) | \( O(1) \) | \( O(\log n) \) | \( O(1) \) \(

† amortized
Extensions of Dijkstra’s algorithm

Dijkstra’s algorithm and proof extend to several related problems:

• Shortest paths in undirected graphs: \( d(v) \leq d(u) + \ell(u, v) \).
• Maximum capacity paths: \( d(v) \geq \min \{ \pi(u), c(u, v) \} \).
• Maximum reliability paths: \( d(v) \geq d(u) \times \gamma(u, v) \).

• ...

Key algebraic structure. Closed semiring (tropical, bottleneck, Viterbi).

4. Greedy Algorithms II

- Dijkstra’s algorithm
- minimum spanning trees
- Prim, Kruskal, Boruvka
- single-link clustering
- min-cost arborescences

Section 6.1

The moral implications of implementing shortest path algorithms

Cycles and cuts

**Def.** A path is a sequence of edges which connects a sequence of nodes.

**Def.** A cycle is a path with no repeated nodes or edges other than the starting and ending nodes.

cycle \( C = \{ (1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 1) \} \)
Cycles and cuts

**Def.** A cut is a partition of the nodes into two nonempty subsets $S$ and $V - S$.

**Def.** The cutset of a cut $S$ is the set of edges with exactly one endpoint in $S$.

![Diagram of a graph with a cut S and cutset D = { (3, 4), (3, 5), (5, 6), (5, 7), (8, 7) }]

Cycle–cut intersection

**Proposition.** A cycle and a cutset intersect in an even number of edges.

![Diagram of a cycle C and cutset D = { (3, 4), (3, 5), (5, 6), (5, 7), (8, 7) } with intersection C ∩ D = { (3, 4), (5, 6) }]

Spanning tree definition

**Def.** Let $H = (V, T)$ be a subgraph of an undirected graph $G = (V, E)$. $H$ is a spanning tree of $G$ if $H$ is both acyclic and connected.

![Diagram of a graph G and spanning tree H = (V, T)]
Spanning tree properties

**Proposition.** Let \( H = (V, T) \) be a subgraph of an undirected graph \( G = (V, E) \). Then, the following are equivalent:

- \( H \) is a spanning tree of \( G \).
- \( H \) is acyclic and connected.
- \( H \) is connected and has \( n-1 \) edges.
- \( H \) is acyclic and has \( n-1 \) edges.
- \( H \) is minimally connected: removal of any edge disconnects it.
- \( H \) is maximally acyclic: addition of any edge creates a cycle.
- \( H \) has a unique simple path between every pair of nodes.

Minimum spanning tree (MST)

**Def.** Given a connected, undirected graph \( G = (V, E) \) with edge costs \( c_{e} \), a minimum spanning tree \( (V, T) \) is a spanning tree of \( G \) such that the sum of the edge costs in \( T \) is minimized.

\[
\text{MST cost} = 50 = 4 + 6 + 8 + 5 + 11 + 9 + 7
\]

Cayley’s theorem. The complete graph on \( n \) nodes has \( n^{n-2} \) spanning trees.

Applications

MST is fundamental problem with diverse applications.

- Dithering.
- Cluster analysis.
- Max bottleneck paths.
- Real-time face verification.
- LDPC codes for error correction.
- Image registration with Renyi entropy.
- Find road networks in satellite and aerial imagery.
- Reducing data storage in sequencing amino acids in a protein.
- Model locality of particle interactions in turbulent fluid flows.
- Autoconfig protocol for Ethernet bridging to avoid cycles in a network.
- Approximation algorithms for NP-hard problems (e.g., TSP, Steiner tree).
- Network design (communication, electrical, hydraulic, computer, road).
**Fundamental cycle**

*Fundamental cycle.* Let \( H = (V, T) \) be a spanning tree of \( G = (V, E) \).

- For any non-tree-edge \( e \in E: T \cup \{ e \} \) contains a unique cycle, say \( C \).
- For any edge \( f \in C: T \cup \{ e \} - \{ f \} \) is a spanning tree.

**Observation.** If \( c_e < c_f \), then \((V, T)\) is not an MST.

![Graph G = (V, E) spanning tree H = (V, T)](image)

**Fundamental cutset**

*Fundamental cutset.* Let \( H = (V, T) \) be a spanning tree of \( G = (V, E) \).

- For any tree edge \( f \in T: T - \{ f \} \) contains two connected components. Let \( D \) be corresponding cutset.
- For any edge \( e \in D: T - \{ f \} \cup \{ e \} \) is a spanning tree.

**Observation.** If \( c_e < c_f \), then \((V, T)\) is not an MST.

![Graph G = (V, E) spanning tree H = (V, T)](image)

**Greedy algorithm: proof of correctness**

**Color invariant.** There exists an MST \((V, T^*)\) containing all of the blue edges and none of the red edges.

**Pf.** [by induction on number of iterations]

**Base case.** No edges colored \( \Rightarrow \) every MST satisfies invariant.

**Red rule.**

- Let \( C \) be a cycle with no red edges.
- Select an uncolored edge of \( C \) of max weight and color it red.

**Blue rule.**

- Let \( D \) be a cutset with no blue edges.
- Select an uncolored edge in \( D \) of min weight and color it blue.

**Greedy algorithm.**

- Apply the red and blue rules (non-deterministically!) until all edges are colored. The blue edges form an MST.
- Note: can stop once \( n - 1 \) edges colored blue.
**Greedy algorithm: proof of correctness**

**Color invariant.** There exists an MST $(V, T^*)$ containing all of the blue edges and none of the red edges.

**Pf.** [by induction on number of iterations]

**Induction step (blue rule).** Suppose color invariant true before blue rule.
- let $D$ be chosen cutset, and let $f$ be edge colored blue.
- if $f \in T^*$, then $T^*$ still satisfies invariant.
- Otherwise, consider fundamental cycle $C$ by adding $f$ to $T^*$.
- let $e \in C$ be another edge in $D$.
- $e$ is uncolored and $c_e \geq c_f$ since
  - $e \in T^* \Rightarrow e$ not red
  - blue rule $\Rightarrow e$ not blue and $c_e \geq c_f$
- Thus, $T^* \cup \{f\} - \{e\}$ satisfies invariant.

**Greedy algorithm: proof of correctness**

**Theorem.** The greedy algorithm terminates. Blue edges form an MST.

**Pf.** We need to show that either the red or blue rule (or both) applies.
- Suppose edge $e$ is left uncolored.
- Blue edges form a forest.
- Case 1: both endpoints of $e$ are in same blue tree.
  $\Rightarrow$ apply red rule to cycle formed by adding $e$ to blue forest.

**Greedy algorithm: proof of correctness**

**Color invariant.** There exists an MST $(V, T^*)$ containing all of the blue edges and none of the red edges.

**Pf.** [by induction on number of iterations]

**Induction step (red rule).** Suppose color invariant true before red rule.
- let $C$ be chosen cycle, and let $e$ be edge colored red.
- if $e \notin T^*$, then $T^*$ still satisfies invariant.
- Otherwise, consider fundamental cutset $D$ by deleting $e$ from $T^*$.
- let $f \in D$ be another edge in $C$.
- $f$ is uncolored and $c_f \geq c_e$ since
  - $f \notin T^* \Rightarrow f$ not blue
  - red rule $\Rightarrow f$ not red and $c_f \geq c_e$
- Thus, $T^* \cup \{f\} - \{e\}$ satisfies invariant.

**Greedy algorithm: proof of correctness**

**Theorem.** The greedy algorithm terminates. Blue edges form an MST.

**Pf.** We need to show that either the red or blue rule (or both) applies.
- Suppose edge $e$ is left uncolored.
- Blue edges form a forest.
- Case 1: both endpoints of $e$ are in same blue tree.
  $\Rightarrow$ apply red rule to cycle formed by adding $e$ to blue forest.
- Case 2: both endpoints of $e$ are in different blue trees.
  $\Rightarrow$ apply blue rule to cutset induced by either of two blue trees.
4. Greedy Algorithms II

- Dijkstra’s algorithm
- minimum spanning trees
- Prim, Kruskal, Boruvka
- single-link clustering
- min-cost arborescences

---

**Prim’s algorithm**

Initialize $S$ = any node, $T$ = $\emptyset$.
Repeat $n - 1$ times:
- Add to $T$ a min-weight edge with one endpoint in $S$.
- Add new node to $S$.

**Theorem.** Prim’s algorithm computes an MST.

**Pf.** Special case of greedy algorithm (blue rule repeatedly applied to $S$). □

---

**Kruskal’s algorithm**

Consider edges in ascending order of weight:
- Add to tree unless it would create a cycle.

**Theorem.** Kruskal’s algorithm computes an MST.

**Pf.** Special case of greedy algorithm.
- Case 1: both endpoints of $e$ in same blue tree.
  - color red by applying red rule to unique cycle.
- Case 2. If both endpoints of $e$ are in different blue trees.
  - color blue by applying blue rule to cutset defined by either tree. □

---

**Prim’s algorithm**: implementation

**Theorem.** Prim’s algorithm can be implemented to run in $O(m \log n)$ time.

**Pf.** Implementation almost identical to Dijkstra’s algorithm.

---

**Kruskal’s algorithm**: implementation

Create an empty priority queue $pq$.

$S \leftarrow \emptyset$, $T \leftarrow \emptyset$.

$s \leftarrow$ any node in $V$.

**FOR** each $v \neq s$:

$\pi[v] \leftarrow \infty$, $\text{pred}[v] \leftarrow \text{null}$; $\pi[s] \leftarrow 0$.

**FOR** each $v \in V$:

**INSERT**($pq$, $v$, $\pi[v]$).

**WHILE** IS-NOT-EMPTY($pq$)

$u \leftarrow \text{DEL-MIN}(pq)$.

$S \leftarrow S \cup \{u\}$, $T \leftarrow T \cup \{\text{pred}(u)\}$.

**FOR** each edge $e = (u, v) \in E$ with $v \notin S$:

If $c_e < \pi[v]$

**DECREASE-KEY**($pq$, $v$, $c_e$).

$\pi[v] \leftarrow c_e$, $\text{pred}[v] \leftarrow e$. 

---
Kruskal’s algorithm: implementation

Theorem. Kruskal’s algorithm can be implemented to run in $O(m \log m)$ time.
- Sort edges by weight.
- Use union-find data structure to dynamically maintain connected components.

**Kruskal** $(V, E, c)$

**SORT** $m$ edges by weight so that $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$.

$T \leftarrow \emptyset$.

**FOREACH** $v \in V$: **MAKE-SET**(v).

**FOR** $i = 1$ to $m$

$(u, v) \leftarrow e_i$.

**IF** **FIND-SET**(u) $\neq$ **FIND-SET**(v)

$T \leftarrow T \cup \{ e_i \}$.

**UNION**(u, v).

**RETURN** $T$.

Reverse-delete algorithm

Consider edges in descending order of weight:
- Remove edge unless it would disconnect the graph.

Theorem. The reverse-delete algorithm computes an MST.

**Pf.** Special case of greedy algorithm.
- Case 1: removing edge $e$ does not disconnect graph.
  - apply red rule to cycle $C$ formed by adding $e$ to existing path between its two endpoints
  - any edge in $C$ with larger weight would have been deleted when considered
- Case 2: removing edge $e$ disconnects graph.
  - apply blue rule to cutset $D$ induced by either component.

**Fact.** [Thorup 2000] Can be implemented to run in $O(m \log n (\log \log n)^3)$ time.

Review: the greedy MST algorithm

**Red rule.**
- Let $C$ be a cycle with no red edges.
- Select an uncolored edge of $C$ of max weight and color it red.

**Blue rule.**
- Let $D$ be a cutset with no blue edges.
- Select an uncolored edge in $D$ of min weight and color it blue.

**Greedy algorithm.**
- Apply the red and blue rules (non-deterministically!) until all edges are colored. The blue edges form an MST.
- Note: can stop once $n-1$ edges colored blue.

Theorem. The greedy algorithm is correct.

Special cases. Prim, Kruskal, reverse-delete, ...

Borůvka’s algorithm

Repeat until only one tree.
- Apply blue rule to cutset corresponding to each blue tree.
- Color all selected edges blue.

Theorem. Borůvka’s algorithm computes the MST.

**Pf.** Special case of greedy algorithm (repeatedly apply blue rule).
Borůvka’s algorithm: implementation

**Theorem.** Borůvka’s algorithm can be implemented to run in $O(m \log n)$ time.

**Pf.**
- To implement a phase in $O(m)$ time:
  - compute connected components of blue edges
  - for each edge $(u, v) \in E$, check if $u$ and $v$ are in different components; if so, update each component’s best edge in cutset
- At most $\log_2 n$ phases since each phase (at least) halves total # trees. 

---

Borůvka’s algorithm on planar graphs

**Theorem.** Borůvka’s algorithm runs in $O(n)$ time on planar graphs.

**Pf.**
- To implement a Borůvka phase in $O(n)$ time:
  - use contraction version of algorithm
  - in planar graphs, $m \leq 3n - 6$.
  - graph stays planar when we contract a blue tree
- Number of nodes (at least) halves.
- At most $\log_2 n$ phases: $cn + cn / 2 + cn / 4 + cn / 8 + \ldots = O(n)$. 

---

Borůvka–Prim algorithm

**Theorem.** The Borůvka–Prim algorithm computes an MST and can be implemented to run in $O(m \log \log n)$ time.

**Pf.**
- Correctness: special case of the greedy algorithm.
- The $\log_2 \log_2 n$ phases of Borůvka’s algorithm take $O(m \log \log n)$ time; resulting graph has at most $n / \log_2 n$ nodes and $m$ edges.
- Prim’s algorithm (using Fibonacci heaps) takes $O(m + n)$ time on a graph with $n / \log_2 n$ nodes and $m$ edges. 

\[
O \left( m + \frac{n}{\log n} \log \left( \frac{n}{\log n} \right) \right)
\]
Does a linear-time MST algorithm exist?

Remark 1. $O(m)$ randomized MST algorithm. [Karger–Klein–Tarjan 1995]
Remark 2. $O(m)$ MST verification algorithm. [Dixon–Rauch–Tarjan 1992]

### Deterministic Compare-based MST Algorithms

<table>
<thead>
<tr>
<th>Year</th>
<th>Worst Case</th>
<th>Discovered By</th>
</tr>
</thead>
<tbody>
<tr>
<td>1975</td>
<td>$O(m \log \log n)$</td>
<td>Yao</td>
</tr>
<tr>
<td>1976</td>
<td>$O(m \log \log n)$</td>
<td>Cheriton–Tarjan</td>
</tr>
<tr>
<td>1984</td>
<td>$O(m \log^* n)$ $O(m + n \log n)$</td>
<td>Fredman–Tarjan</td>
</tr>
<tr>
<td>1986</td>
<td>$O(m \log (\log^* n))$</td>
<td>Gabow–Galil–Spencer–Tarjan</td>
</tr>
<tr>
<td>1997</td>
<td>$O(m \alpha(n) \log \alpha(n))$</td>
<td>Chazelle</td>
</tr>
<tr>
<td>2000</td>
<td>$O(m \alpha(n))$</td>
<td>Chazelle</td>
</tr>
<tr>
<td>2002</td>
<td>Asymptotically optimal</td>
<td>Pettie–Ramachandran</td>
</tr>
<tr>
<td>20xx</td>
<td>$O(m)$</td>
<td>???</td>
</tr>
</tbody>
</table>

4. **Greedy Algorithms II**

- Dijkstra’s algorithm
- Minimum spanning trees
- Prim, Kruskal, Boruvka
- Single-link clustering
- Min-cost arborescences

#### Clustering

**Goal.** Given a set $U$ of $n$ objects labeled $p_1, ..., p_n$, partition into clusters so that objects in different clusters are far apart.

**Applications.**

- Routing in mobile ad hoc networks.
- Document categorization for web search.
- Similarity searching in medical image databases
- Skycat: cluster $10^9$ sky objects into stars, quasars, galaxies.
- ...

**Clustering of Maximum Spacing**

**k-clustering.** Divide objects into $k$ non-empty groups.

**Distance function.** Numeric value specifying “closeness” of two objects.

- $d(p_i, p_j) = 0$ iff $p_i = p_j$ [identity of indiscernibles]
- $d(p_i, p_j) \geq 0$ [non-negativity]
- $d(p_i, p_j) = d(p_j, p_i)$ [symmetry]

**Spacing.** Min distance between any pair of points in different clusters.

**Goal.** Given an integer $k$, find a $k$-clustering of maximum spacing.
Greedy clustering algorithm

"Well-known" algorithm in science literature for single-linkage k-clustering:

- Form a graph on the node set $U$, corresponding to $n$ clusters.
- Find the closest pair of objects such that each object is in a different cluster, and add an edge between them.
- Repeat $n - k$ times until there are exactly $k$ clusters.

**Key observation.** This procedure is precisely Kruskal’s algorithm (except we stop when there are $k$ connected components).

**Alternative.** Find an MST and delete the $(k - 1)$ longest edges.

---

**Greedy clustering algorithm: analysis**

Theorem. Let $C^*$ denote the clustering $C^*_1, \ldots, C^*_k$ formed by deleting the $k - 1$ longest edges of an MST. Then, $C^*$ is a $k$-clustering of max spacing.

**Pf.** Let $C$ denote some other clustering $C_1, \ldots, C_k$.

- The spacing of $C^*$ is the length $d^*$ of the $(k - 1)$th longest edge in MST.
- Let $p_i$ and $p_j$ be in the same cluster in $C^*$, say $C^*_r$, but different clusters in $C$, say $C_s$ and $C_t$.
- Some edge $(p, q)$ on $p_i - p_j$ path in $C^*$, spans two different clusters in $C$.
- Edge $(p, q)$ has length $\leq d^*$ since it wasn’t deleted.
- Spacing of $C$ is $\leq d^*$ since $p$ and $q$ are in different clusters.

---

Dendrogram of cancers in human

Tumors in similar tissues cluster together.

---

4. **Greedy Algorithms II**

- Dijkstra’s algorithm
- minimum spanning trees
- Prim, Kruskal, Boruvka
- single-link clustering
- min-cost arborescences
**Arborescences**

**Def.** Given a digraph $G = (V, E)$ and a root $r \in V$, an arborescence (rooted at $r$) is a subgraph $T = (V, F)$ such that
- $T$ is a spanning tree of $G$ if we ignore the direction of edges.
- There is a directed path in $T$ from $r$ to each other node $v \in V$.

**Warmup.** Given a digraph $G$, find an arborescence rooted at $r$ (if one exists).

**Algorithm.** BFS or DFS from $r$ is an arborescence (iff all nodes reachable).

**Min-cost arborescence problem**

**Problem.** Given a digraph $G$ with a root node $r$ and with a nonnegative cost $c_e \geq 0$ on each edge $e$, compute an arborescence rooted at $r$ of minimum cost.

**Assumption 1.** $G$ has an arborescence rooted at $r$.

**Assumption 2.** No edge enters $r$ (safe to delete since they won’t help).

**Simple greedy approaches do not work**

**Observations.** A min-cost arborescence need not:
- Be a shortest-paths tree.
- Include the cheapest edge (in some cut).
- Exclude the most expensive edge (in some cycle).
A sufficient optimality condition

**Property.** For each node $v \neq r$, choose one cheapest edge entering $v$ and let $F^*$ denote this set of $n-1$ edges. If $(V, F^*)$ is an arborescence, then it is a min-cost arborescence.

**Pf.** An arborescence needs exactly one edge entering each node $v \neq r$ and $(V, F^*)$ is the cheapest way to make these choices. •

---

Reduced costs

**Def.** For each $v \neq r$, let $y(v)$ denote the min cost of any edge entering $v$. The **reduced cost** of an edge $(u, v)$ is $c'(u, v) = c(u, v) - y(v) \geq 0$.

**Observation.** $T$ is a min-cost arborescence in $G$ using costs $c$ iff $T$ is a min-cost arborescence in $G$ using reduced costs $c'$.

**Pf.** Each arborescence has exactly one edge entering $v$.

---

Edmonds branching algorithm: intuition

**Intuition.** Recall $F^*$ is a set of cheapest edges entering $v$ for each $v \neq r$.

- Now, all edges in $F^*$ have 0 cost with respect to costs $c'(u, v)$.
- If $F^*$ does not contain a cycle, then it is a min-cost arborescence.
- If $F^*$ contains a cycle $C$, can afford to use as many edges in $C$ as desired.
  - Contract nodes in $C$ to a supernode (removing any self-loops).
  - Recursively solve problem in contracted network $G'$ with costs $c'(u, v)$.
Edmonds branching algorithm: intuition

**Intuition.** Recall $F^* = \text{set of cheapest edges entering } v$ for each $v \neq r$.
- Now, all edges in $F^*$ have 0 cost with respect to costs $c(u, v)$.
- If $F^*$ does not contain a cycle, then it is a min-cost arborescence.
- If $F^*$ contains a cycle $C$, can afford to use as many edges in $C$ as desired.
- Contract nodes in $C$ to a supernode (removing any self-loops).
- Recursively solve problem in contracted network $G'$ with costs $c'(u, v)$.

---

**Edmonds branching algorithm**

**Edmonds-BRANCHING** $(G, r, c)$

**FOREACH** $v \neq r$

\[ y(v) \leftarrow \min \text{ cost of an edge entering } v. \]

\[ c'(u, v) \leftarrow c'(u, v) - y(v) \text{ for each edge } (u, v) \text{ entering } v. \]

**FOREACH** $v \neq r$: choose one 0-cost edge entering $v$ and let $F^*$ be the resulting set of edges.

- **IF** $F^*$ forms an arborescence, **RETURN** $T = (V, F^*)$.
- **ELSE**
  - $C \leftarrow$ directed cycle in $F^*$.
  - Contract $C$ to a single supernode, yielding $G' = (V', E')$.
  - $T' \leftarrow$ **Edmonds-BRANCHING** $(G', r, c')$
  - **Extend** $T'$ to an arborescence $T$ in $G$ by adding all but one edge of $C$.
  - **RETURN** $T$.

---

**Edmonds branching algorithm: key lemma**

**Q.** What could go wrong?

**A.**
- Min-cost arborescence in $G'$ has exactly one edge entering a node in $C$ (since $C$ is contracted to a single node)
- But min-cost arborescence in $G$ might have more edges entering $C$.

---

**Edmonds branching algorithm**

**Q.** Can something go wrong?

**A.**
- Min-cost arborescence in $G'$ has exactly one edge entering a node in $C$ (since $C$ is contracted to a single node)
- But min-cost arborescence in $G$ might have more edges entering $C$.

---

**Lemma.** Let $C$ be a cycle in $G$ consisting of 0-cost edges. There exists a min-cost arborescence rooted at $r$ that has exactly one edge entering $C$.

**Proof.** Let $T$ be a min-cost arborescence rooted at $r$.

- **Case 0.** $T$ has no edges entering $C$.
  - Since $T$ is an arborescence, there is an $r \rightarrow v$ path fore each node $v$.
  - It implies at least one edge enters $C$.\[ \Rightarrow \]

- **Case 1.** $T$ has exactly one edge entering $C$.
  - $T$ satisfies the lemma.

- **Case 2.** $T$ has more than one edge that enters $C$.
  - We construct another min-cost arborescence $T'$ that has exactly one edge entering $C$.\[ \Rightarrow \]
Edmonds branching algorithm: key lemma

Case 2 construction of $T'$.
- Let $(a, b)$ be an edge in $T$ entering $C$ that lies on a shortest path from $r$.
- We delete all edges of $T$ that enter a node in $C$ except $(a, b)$.
- We add in all edges of $C$ except the one that enters $b$.

Claim. $T'$ is a min-cost arborescence.
- The cost of $T'$ is at most that of $T$ since we add only 0-cost edges.
- $T'$ has exactly one edge entering each node $v \neq r$.
- $T'$ has no directed cycles.

Edmonds branching algorithm: analysis


Pf. [by induction on number of nodes in $G$]
- If the edges of $F^*$ form an arborescence, then min-cost arborescence.
- Otherwise, we use reduced costs, which is equivalent.
- After contracting a 0-cost cycle $C$ to obtain a smaller graph $G'$, the algorithm finds a min-cost arborescence $T'$ in $G'$ (by induction).
- Key lemma: there exists a min-cost arborescence $T$ in $G$ that corresponds to $T'$.

Theorem. The greedy algorithm can be implemented to run in $O(mn)$ time.
Pf.
- At most $n$ contractions (since each reduces the number of nodes).
- Finding and contracting the cycle $C$ takes $O(m)$ time.
- Transforming $T'$ into $T$ takes $O(m)$ time.

Min-cost arborescence

Theorem. [Gabow–Galil–Spencer–Tarjan 1985] There exists an $O(m + n \log n)$ time algorithm to compute a min-cost arborescence.

Min-cost arborescence

Theorem. [Gabow–Galil–Spencer–Tarjan 1985] There exists an $O(m + n \log n)$ time algorithm to compute a min-cost arborescence.