### 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 $t$.

![Graph](image)

**Solution.**

```
 length of path = 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.

![Graph](image)

**Solution.**

```
shortest-paths tree
```
Car navigation

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

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 \( u \in S \), \( d(u) \) is the length of a shortest \( s \to u \) path.

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

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

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

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

\[
\ell(P) \geq \ell(P') + \ell(x, y) \geq d(x) + \ell(x, y) \geq \pi(y) \geq \pi(v) \]

**Dijkstra’s algorithm: efficient implementation**

**Critical optimization 1.** For each unexplored node \( v \), explicitly maintain \( \pi(v) \) instead of computing directly from formula:

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

- For each \( v \not\in S \), \( \pi(v) \) can only decrease (because \( S \) only increases).
- More specifically, suppose \( u \) is added to \( S \) and there is an edge \((u, v)\) leaving \( u \). Then, it suffices to update:

\[
\pi(v) = \min \{ \pi(v), d(u) + \ell(u, v) \}
\]

**Critical optimization 2.** Use a priority queue to choose an unexplored node that minimizes \( \pi(v) \).

**Dijkstra’s algorithm: implementation**

**Implementation.**

- Algorithm stores \( \pi(v) \) for each node \( v \).
- Priority queue stores \( \pi(v) \) for each unexplored node \( v \).
- Recall that \( \pi(v) = d(v) \) once vertex is deleted from priority queue.

[Dijkstra algorithm pseudocode]

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

**Performance.** Depends on priority queue: \( n \) INSERT, \( n \) DELETE-MIN, \( m \) DECREASE-KEY.

- Array implementation optimal for dense graphs.
- Binary heap much faster for sparse graphs.
- 4-way heap worth the trouble in performance-critical situations.
- Fibonacci/Brodal best in theory, but 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(d \log_d n) )</td>
<td>( O(m \log_{\log_m} n) )</td>
</tr>
<tr>
<td>Fibonacci heap (Fredman–Tarjan 1984)</td>
<td>( O(1) )</td>
<td>( O(\log n) )</td>
<td>( O(1) )</td>
<td>( O(m + n \log n) )</td>
</tr>
<tr>
<td>Brodal queue (Brodal 1996)</td>
<td>( O(1) )</td>
<td>( O(\log n) )</td>
<td>( O(1) )</td>
<td>( O(m + n \log n) )</td>
</tr>
</tbody>
</table>

\( \dagger \) 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

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 $C = \{ (1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 1) \}$

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.

\[
\text{cutset } D = \{ (3, 4), (3, 5), (5, 6), (5, 7), (8, 7) \} \\
\text{cycle } C = \{ (1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 1) \} \\
\text{intersection } C \cap D = \{ (3, 4), (5, 6) \}
\]

**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.

\( H = (V, T) \) is a spanning tree of \( G = (V, E) \)
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.

![Spanning tree example](image)

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.

![MST example](image)

Cayley’s theorem. There are $n^{n-2}$ spanning trees of complete graph on $n$ vertices. ❯ can’t solve by brute force

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 $(V, T)$ be a spanning tree of $G = (V, E)$.

- Adding any non-tree edge $e \in E$ to $T$ forms unique cycle $C$.
- Deleting any edge $f \in C$ from $T \cup \{e\}$ results in a spanning tree.

![Fundamental cycle example](image)

Observation. If $c_e < c_f$, then $(V, T)$ is not an MST.
Fundamental cutset. Let \((V, T)\) be a spanning tree of \(G = (V, E)\).
- Deleting any tree edge \(f\) from \(T\) divides nodes of spanning tree into two connected components. Let \(D\) be cutset.
- Adding any edge \(e \in D \to T \setminus \{f\}\) results in a spanning tree.

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

The greedy 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.

**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.

**Induction step (blue rule).** Suppose color invariant true before blue rule.
- let \(D\) be chosen cutset, and let \(f\) be edge colored blue.
- if \(f \in T^*, T^*\) 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
  - \(e \not\in T^* \Rightarrow e\) not red
  - blue rule \(\Rightarrow e\) not blue and \(c_e \geq c_f\)
- Thus, \(T^* \cup \{f\} \setminus \{e\}\) satisfies invariant.
**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 \(e \notin T^*\), \(T^*\) 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 \(c_e \geq c_f\) since
  - \(f \notin T^*\) \(\Rightarrow\) \(f\) not blue
  - red rule \(\Rightarrow\) \(f\) not red 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.
- 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

**Section 6.2**
**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.  
  $\Rightarrow$ color red by applying red rule to unique cycle.  
- Case 2. If both endpoints of $e$ are in different blue trees.  
  $\Rightarrow$ 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.

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

1. Create an empty priority queue $pq$.
2. $s \leftarrow$ any node in $V$.
3. FOREACH $v \neq s$ : $\pi(v) \leftarrow \infty$; $\pi(s) \leftarrow 0$.
4. FOREACH $v \in V$ : INSERT($pq, v, \pi(v)$).
5. WHILE (IS-NOT-EMPTY($pq$))
   - $u \leftarrow$ DEL-MIN($pq$).
   - $T \leftarrow T \cup$ pred($u$).
   - FOREACH edge $(u, v) \in E$ incident to $u$:
     - IF $\pi(v) > c(u, v)$
       - DECREASE-KEY($pq, v, c(u, v)$).
       - $\pi(v) \leftarrow c(u, v)$; pred($v$) $\leftarrow (u, v)$.

---

**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** ($V, E$) 

1. SORT $m$ edges by weight so that $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$.  
2. $T \leftarrow \emptyset$.
3. FOREACH $v \in V$ : MAKE-SET($v$).
4. 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($u, v$).  
5. RETURN $T$.  

---

**Diagram:**

- Prim’s algorithm: Initial configuration.  
- Prim’s algorithm: Incremental build-up.  
- Kruskal’s algorithm: Initial edge selection.  
- Kruskal’s algorithm: Incremental build-up.
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.
  - $\Rightarrow$ 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.
  - $\Rightarrow$ 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: implementation

Node contraction version.
- After each phase, contract each blue tree to a single supernode.
- Delete parallel edges (keeping only cheapest one) and self loops.
- Borůvka phase becomes: take cheapest edge incident to each node.

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

- Run Borůvka (contraction version) for \( \log_2 \log_2 n \) phases.
- Run Prim on resulting, contracted graph.

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.

Does a linear-time MST algorithm exist?

<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>optimal</td>
<td>Pettie–Ramachandran</td>
</tr>
<tr>
<td>20xx</td>
<td>( O(m) )</td>
<td>??</td>
</tr>
</tbody>
</table>

Remark 1. \( O(m) \) randomized MST algorithm. [Karger–Klein–Tarjan 1995]

Remark 2. \( O(m) \) MST verification algorithm. [Dixon–Rauch–Tarjan 1992]
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 $p_1, \ldots, p_n$, 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 $p_i = p_j$ \hspace{1cm} [identity of indiscernibles]
- $d(p_i, p_j) \geq 0$ \hspace{1cm} [non-negativity]
- $d(p_i, p_j) = d(p_j, p_i)$ \hspace{1cm} [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 \( C^*_1, ..., C^*_k \) 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 \( C_1, ..., C_k \).

- 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.

Reference: Botstein & Brown group

gene I

gene I expressed

gene I not expressed

dendrogram

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).
**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 $v \in V$.

**Proposition.** A subgraph $T = (V, F)$ of $G$ is an arborescence rooted at $r$ iff $T$ has no directed cycles and each node $v \neq r$ has exactly one entering edge.

**Pf.**

⇒ If $T$ is an arborescence, then no (directed) cycles and every node $v \neq r$ has exactly one entering edge—the last edge on the unique $r \rightarrow v$ path.

⇐ Suppose $T$ has no cycles and each node $v \neq r$ has one entering edge.
- To construct an $r \rightarrow v$ path, start at $v$ and repeatedly follow edges in the backward direction.
- Since $T$ has no directed cycles, the process must terminate.
- It must terminate at $r$ since $r$ is the only node with no entering edge. □

---

**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.

---

**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 $F^v$ denote this set of $n-1$ edges. If $(V, F^v)$ is an arborescence, then it is a min-cost arborescence.

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

**Note.** \( F^* \) may not be an arborescence (since it may have directed cycles).

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 \( c'(u, v) \).

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 \( F^* = \text{set of cheapest edges entering } v \) 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** \((G, r, c)\)

**FOR** \(v \neq r\)
- \(y(v) \leftarrow \text{min cost of an edge entering } v\).
- \(c'(u, v) \leftarrow c'(u, v) - y(v)\) for each edge \((u, v)\) entering \(v\).

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

**IF** \(F^*\) forms an arborescence, \(\text{RETURN } T = (V, F^*)\).

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

Edmonds branching algorithm: key lemma

**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 \(C\).

**Pf.** 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 for each node \(v \Rightarrow\) at least one edge enters \(C\).

**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 \(C\).

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 \(b\).
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.

(The graph had no cycles before; no cycles within $C$; now only $(a, b)$ enters $C$)

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.