Algorithm II

4. Greedy Algorithms II

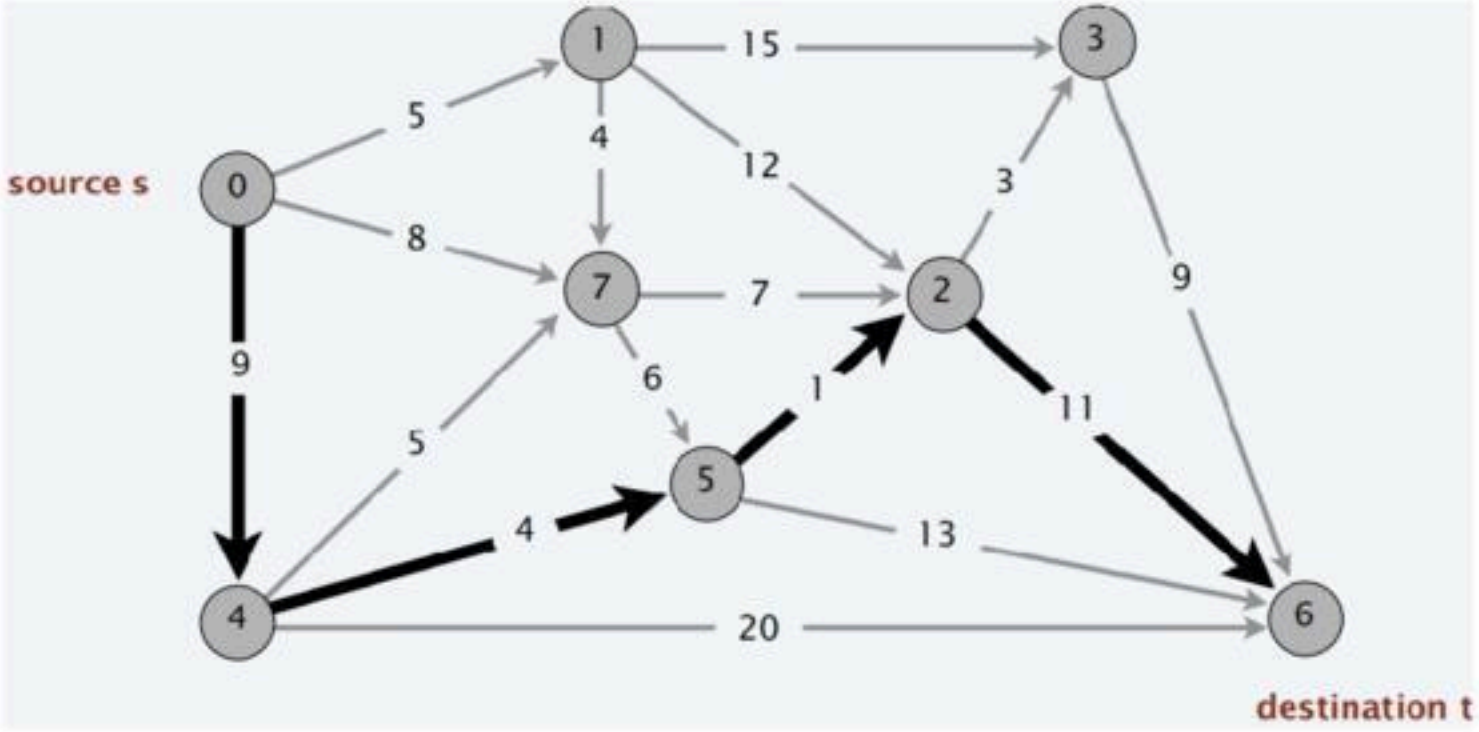
WU Xiaokun 吴晓堃

xkun.wu [at] gmail

Dijkstra's algorithm

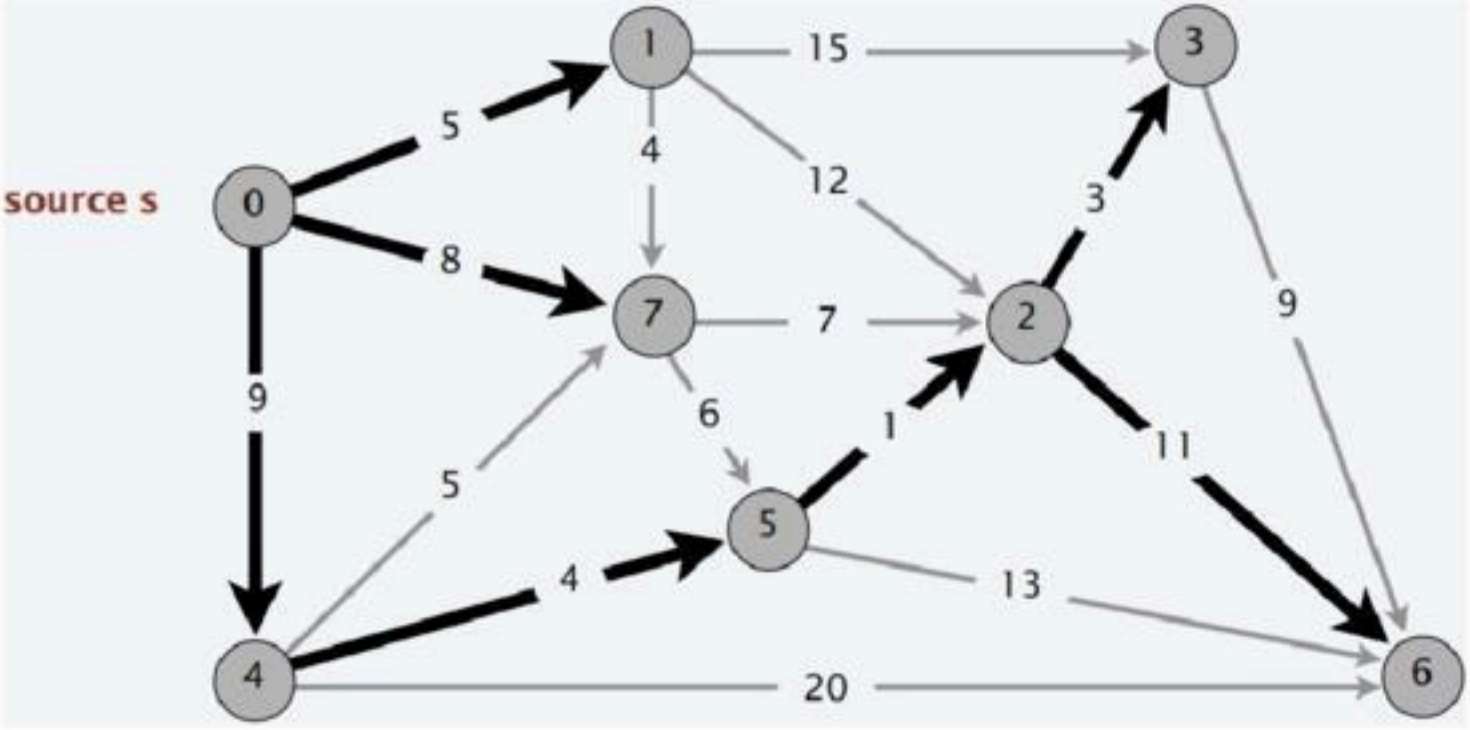
Single-pair shortest path problem

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



Single-source shortest paths problem

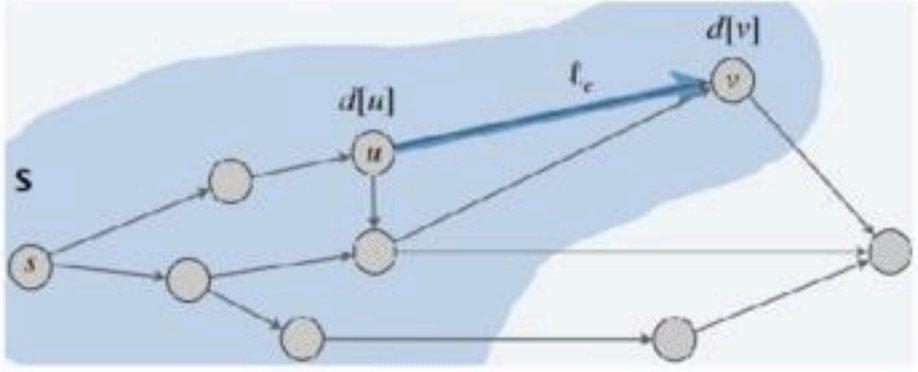
Problem. Given a digraph $G = (V, E)$, edge lengths $l_e \geq 0$, source $s \in V$, find a shortest directed path from s to every node.



Dijkstra's (single-source shortest-paths)

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

- Initialize $S = \{s\}$, $d[s] = 0$.
- Repeatedly choose unexplored node $v \notin S$ which minimizes $\pi(v) = \min_{e=(u,v):u \in S} d[u] + l_e$.
  - add v to S, and set $d[v] = \pi(v)$.
- To recover path, set $pred[v]$ = e that achieves `min`.



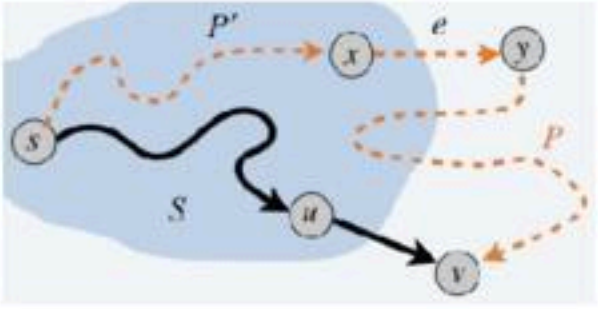
Dijkstra's: proof of correctness

Invariant. For each node $u \notin S$: $d[u]$ = length of a shortest $s \rightsquigarrow u$ path.

Pf. [by induction on $|S|$]

$|S| = 1$ is clear: $S = \{s\}$, $d[s] = 0$; now assume true for $|S| \geq 1$.

- Let v be next node added to S, and let (u, v) be the last edge.
  - A shortest $s \rightsquigarrow u$ path plus (u, v) is an $s \rightsquigarrow v$ path of length $\pi(v)$.
- Consider any other $s \rightsquigarrow v$ path P: show it's no shorter than $\pi(v)$.
  - Let $e = (x, y) \in P$ leaves S first; P' the subpath $s \rightsquigarrow x$.
  - Length of P is already $\geq \pi(v)$ when reaches y:
    - $l(P) \geq l(P') + l_e \geq d[x] + l_e \geq \pi(y) \geq \pi(v)$.



Dijkstra's: efficient implementation

Critical optimization 1. For each unexplored node $v \notin S$: explicitly maintain $\pi[v]$ instead of computing directly from definition $\pi(v) = \min_{e=(u,v):u \in S} d[u] + l_e$

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

Dijkstra's: efficient implementation

Critical optimization 1. For each unexplored node $v \notin S$: explicitly maintain $\pi[v]$ instead of computing directly from definition $\pi(v) = \min_{e=(u,v):u \in S} d[u] + l_e$

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

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

Dijkstra's: efficient implementation

Critical optimization 1. For each unexplored node $v \notin S$: explicitly maintain $\pi[v]$ instead of computing directly from definition $\pi(v) = \min_{e=(u,v):u \in S} d[u] + l_e$

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

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

Implementation.

- Algorithm maintains $\pi[v]$ for each node v.
- Priority queue stores unexplored nodes, using $\pi[\cdot]$ as priorities.
- Once u is deleted from the PQ, $\pi[u]$ = length of a shortest $s \rightsquigarrow u$ path.

Dijkstra's: algorithm

1. FOREACH $v \neq s$: $\pi[v] = \infty$, $pred[v]$ = `null`; $\pi[s] = 0$;
2. Create an empty priority queue `pq`;
3. FOREACH $v \in V$: `INSERT`(`pq`, v, $\pi[v]$);
4. WHILE (IS-NOT-EMPTY(`pq`)):
    1. u = `DEL-MIN`(`pq`);
    2. FOREACH edge $e = (u, v) \in E$ leaving u:
        1. IF ($\pi[v] \leq \pi[u] + l_e$): CONTINUE;
        2. `DECREASE-KEY`(`pq`, v, $\pi[u] + l_e$);
        3. $\pi[v] = \pi[u] + l_e$;
        4. $pred[v] = e$;

Demo: Dijkstra's algorithm

Dijkstra's: analysis

Performance. Depends on PQ: n INSERT, n DELETE-MIN, $\leq m$ DECREASE-KEY.

- each priority queue operation can be made to run in $O(\log n)$ time.
- overall time for the implementation is $O(m \log n)$.

Dijkstra's: which priority queue?

Performance. Depends on PQ: n INSERT, n DELETE-MIN, $\leq m$ DECREASE-KEY.

- Array implementation optimal for dense graphs ($\Theta(n^2)$ edges).
- Binary heap much faster for sparse graphs ($\Theta(n)$ edges).
- 4-way heap worth the trouble in performance-critical situations.

priority queue	INSERT	DELETE-MIN	DECREASE-KEY	total
node-indexed array (A[i] = priority of i)	$O(1)$	$O(n)$	$O(1)$	$O(n^2)$
binary heap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(m \log n)$
d-way heap (Johnson 1975)	$O(d \log_d n)$	$O(d \log_d n)$	$O(\log_d n)$	$O(m \log_{m/n} n)$
Fibonacci heap (Fredman-Tarjan 1984)	$O(1)$	$O(\log n)$ †	$O(1)$ †	$O(m + n \log n)$
integer priority queue (Thorup 2004)	$O(1)$	$O(\log \log n)$	$O(1)$	$O(m + n \log \log n)$

Quiz: single-source shortest-paths

How to solve the the single-source shortest paths problem in *undirected* graphs with positive edge lengths?

A. Replace each undirected edge with two antiparallel edges of same length. Run Dijkstra's algorithm in the resulting digraph.

B. Modify Dijkstra's algorithms so that when it processes node u, it consider all edges incident to u (instead of edges leaving u).

C. Either A or B.

D. Neither A nor B.

Quiz: single-source shortest-paths

How to solve the the single-source shortest paths problem in *undirected* graphs with positive edge lengths?

A. Replace each undirected edge with two antiparallel edges of same length. Run Dijkstra's algorithm in the resulting digraph.

B. Modify Dijkstra's algorithms so that when it processes node u, it consider all edges incident to u (instead of edges leaving u).

C. Either A or B.

D. Neither A nor B.

- A is standard treatment.
- B also works.

Undirected single-source shortest paths

Theorem. [Thorup 1999] Can solve single-source shortest paths problem in undirected graphs with positive integer edge lengths in $O(m)$ time.

Remark. Does not explore nodes in increasing order of distance from s.

Dijkstra's: extensions

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

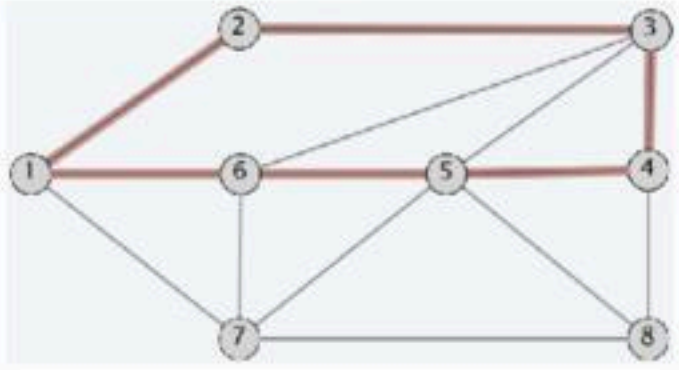
- Shortest paths in undirected graphs: $\pi[v] \leq \pi[u] + l(u, v)$.
- Maximum capacity paths: $\pi[v] \geq min\{\pi[u], c(u, v)\}$.
- Maximum reliability paths: $\pi[v] \geq \pi[u] \times \gamma(u, v)$

Minimum Spanning Tree

Cycles

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.

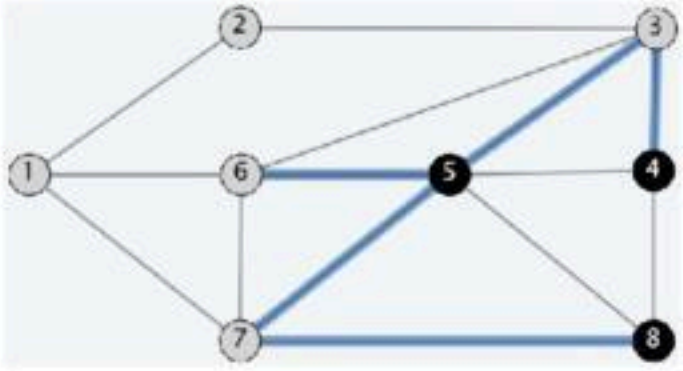


- path P = { (1, 2), (2, 3), (3, 4), (4, 5), (5, 6) }
- cycle C = { (1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 1) }

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.

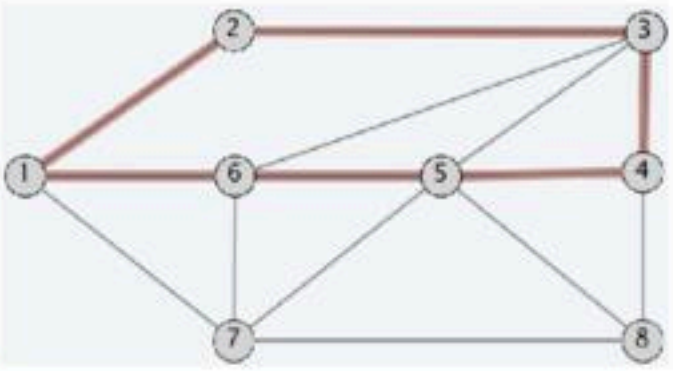


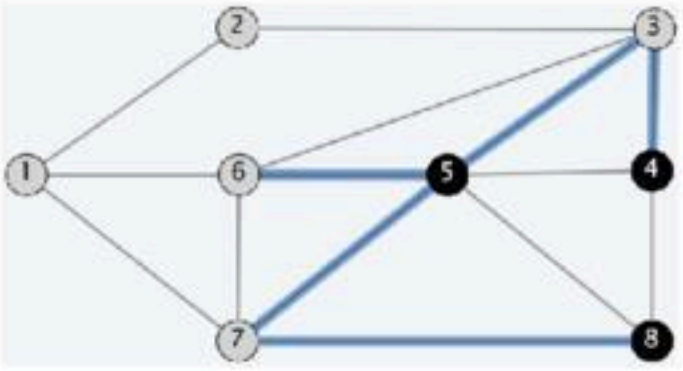
- cut S = { 4, 5, 8 }
- cutset D = { (3, 4), (3, 5), (5, 6), (5, 7), (8, 7) }

Quiz: Cutset

Let C be a cycle and let D be a cutset. How many edges do C and D have in common? Choose the best answer.

- 0
- 2
- not 1
- an even number

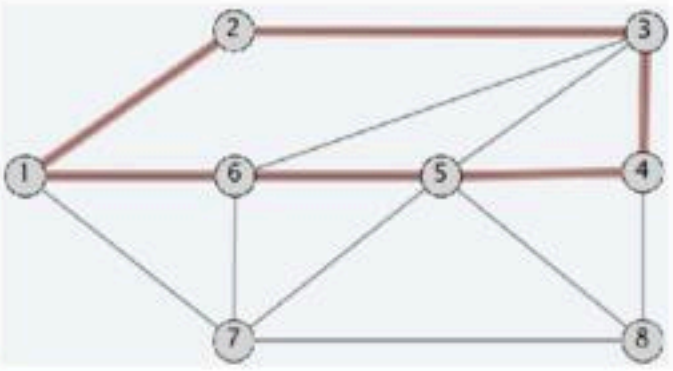


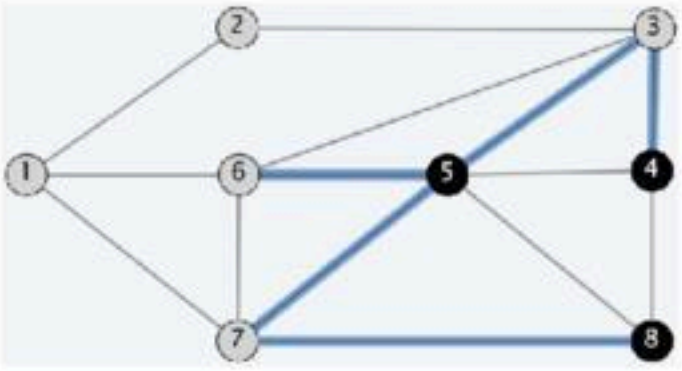


Quiz: Cutset

Let C be a cycle and let D be a cutset. How many edges do C and D have in common? Choose the best answer.

- 0
- 2
- not 1
- an even number

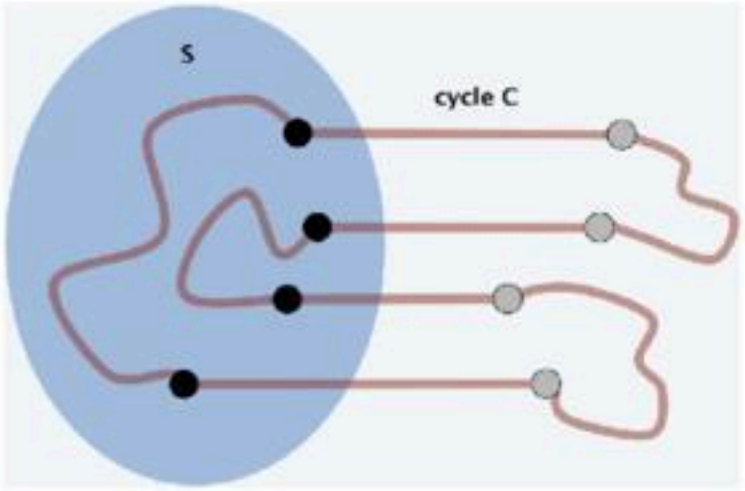




Hint: cycle cross the cut, so even number of times

Cycle–cut intersection

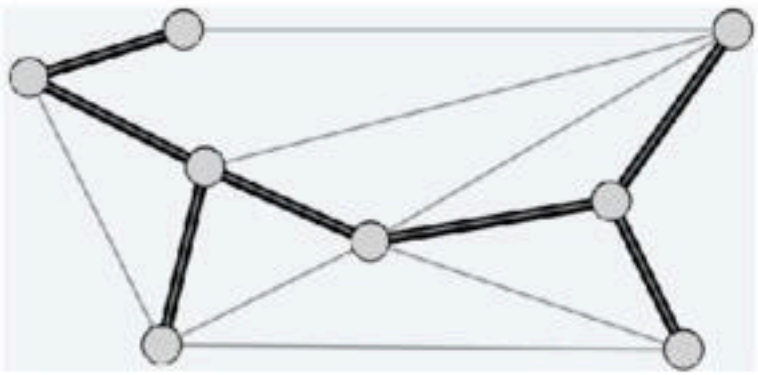
Proposition. A cycle and a cutset intersect in an even number of edges.



Spanning tree

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*.

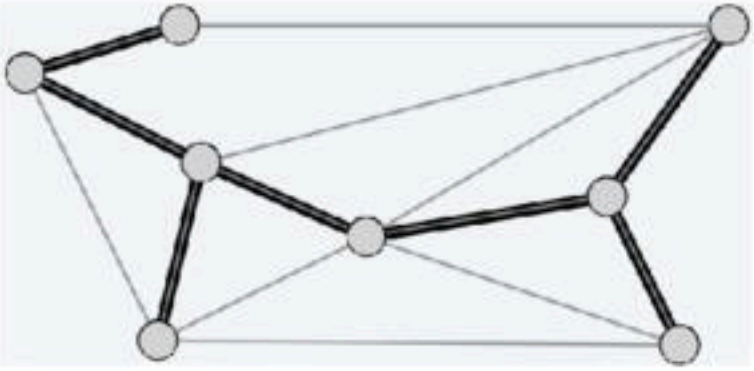
- the minimum effort to connect vertices topologically



Quiz: spanning tree

Which of the following properties are true for all spanning trees H?

- Contains exactly $|V|-1$ edges.
- The removal of any edge disconnects it.
- The addition of any edge creates a cycle.
- All of the above.

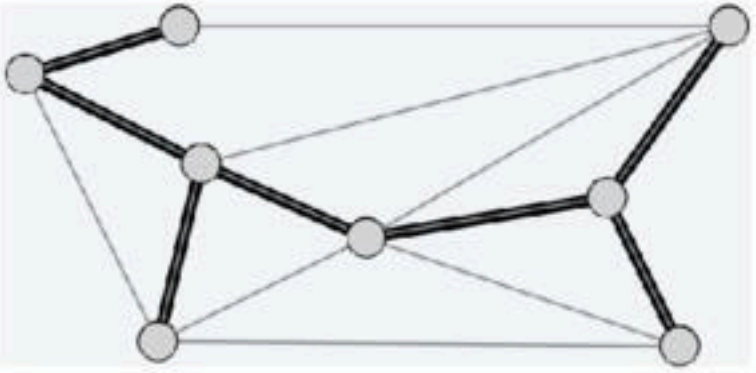


Spanning tree: properties

Proposition. Let $H = (V, T)$ be a subgraph of an undirected graph $G = (V, E)$.

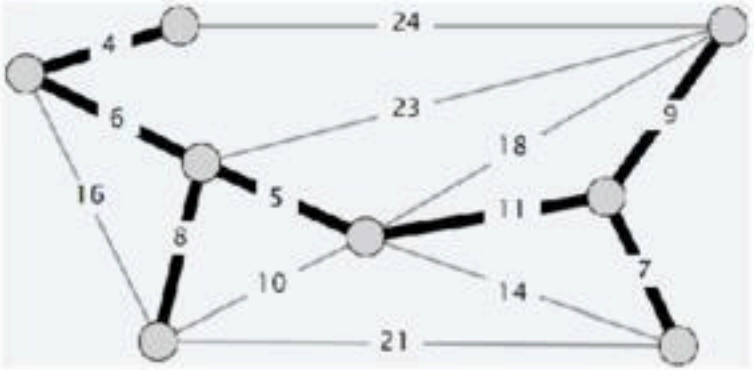
Then, the following are equivalent:

- H is a spanning tree of G.
- H is acyclic and connected.
- H is connected and has $|V| - 1$ edges.
- H is acyclic and has $|V| - 1$ edges.
- H is minimally connected: removal of any edge disconnects it.
- H is maximally acyclic: addition of any edge creates a cycle.



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.



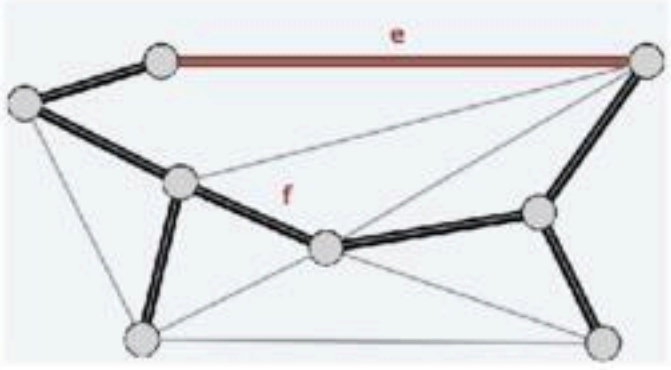
Cayley's theorem. The complete graph on n nodes has n^{n-2} spanning trees.

- can't solve by brute-force

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 $f \in C : (V, T \cup \{e\}–\{f\})$ is a spanning tree.

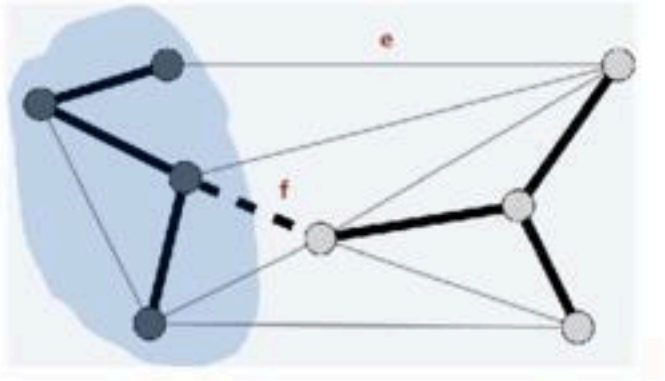


Observation. If $c_e < c_f$, then (V, T) is not an MST.

Fundamental cutset

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

- For any tree edge $f \in T$: $(V, T - \{f\})$ has two connected components.
  - Let D denote corresponding cutset.
- For any edge $e \in D$: $(V, T - f \cup e)$ is a spanning tree.



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

MST: greedy coloring

Red rule.

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

Blue rule.

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

MST: greedy coloring

Red rule.

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

Blue rule.

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

Greedy coloring.

- 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.

Demo: Greedy coloring

Greedy coloring: invariant

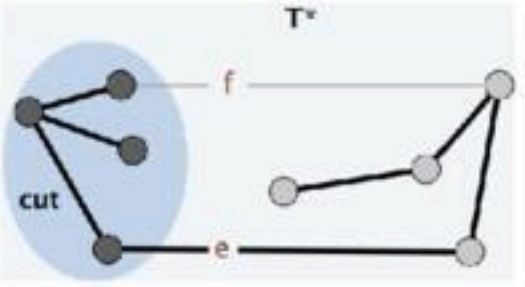
Color invariant. There exists an $MST(V, T^*)$ containing every blue edge and no red edge.

Pf. 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 coloring: invariant (cont.)

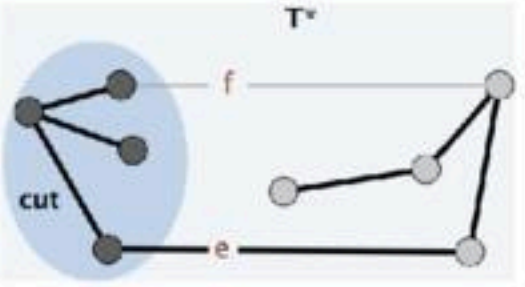
Color invariant. There exists an $MST(V, T^*)$ containing every blue edge and no red edge.

Pf. 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 $c_e \geq c_f$ 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.

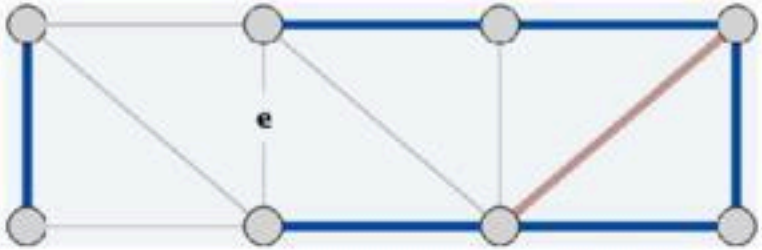


Greedy coloring: correctness

Theorem. The greedy coloring algorithm terminates. Blue edges form an MST.
Pf. show that either red or blue rule (or both) applies in each step.

Blue edges keep growing until forming a forest.

- Suppose edge e is left uncolored.
- Case 1: both endpoints of e are in same blue tree.
  - apply red rule to cycle formed by adding e to blue forest.

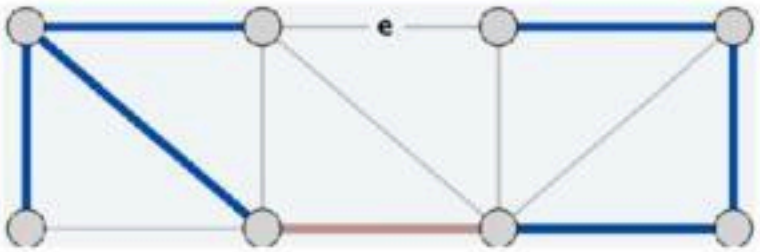


Greedy coloring: correctness (cont.)

Theorem. The greedy coloring algorithm terminates. Blue edges form an MST.
Pf. show that either red or blue rule (or both) applies in each step.

Blue edges keep growing until forming a forest.

- Suppose edge e is left uncolored.
- Case 1: both endpoints of e are in same blue tree.
  - apply red rule to cycle formed by adding e to blue forest.
- Case 2: endpoints of e are in different blue trees.
  - apply blue rule to cutset induced by either of two blue trees.



Prim, Kruskal, Boruvka

Prim's algorithm

Initialize $S = \{s\}$ for any node s, $T = \emptyset$.

Repeat $n-1$ times:

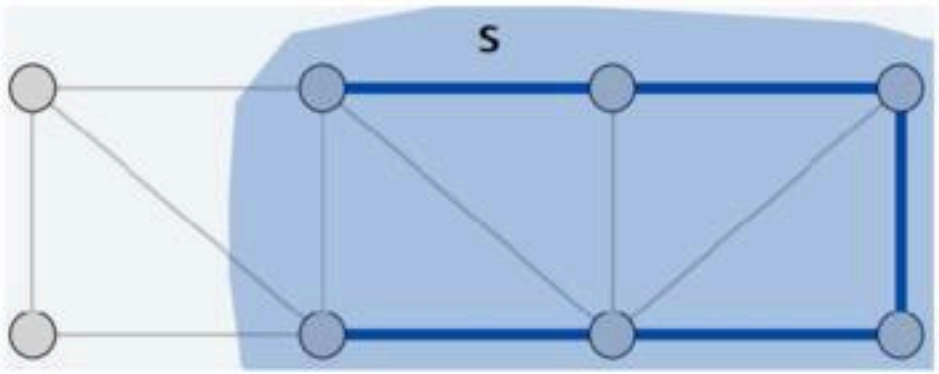
- Add to T a min-cost edge with exactly one endpoint in S.
- Add the other endpoint to S.

Prim's algorithm

Initialize $S = \{s\}$ for any node s, $T = \emptyset$.
Repeat $n-1$ times:

- Add to T a min-cost edge with exactly one endpoint in S.
- Add the other endpoint to S.

Theorem. Prim's algorithm computes an MST.
Pf. Special case of greedy coloring (blue rule repeatedly applied to S).



Prim's algorithm: implementation

1. $S = \emptyset, T = \emptyset$;
2. s = any node in V;
3. FOREACH $v \neq s$: $\pi[v] = \infty$, $pred[v]$ = `null`; $\pi[s] = 0$;
4. Create an empty priority queue `pq`;
5. FOREACH $v \in V$: `INSERT`(`pq`, v, $\pi[v]$);
6. WHILE (IS-NOT-EMPTY(`pq`)):
    1. u = `DEL-MIN`(`pq`);
    2. $S = S \cup u, T = T \cup \{pred[u]\}$;
    3. FOREACH edge $e = (u, v) \in E$ with $v \notin S$:
        1. IF ($c_e \geq \pi[v]$): CONTINUE;
        2. `DECREASE-KEY`(`pq`, v, c_e);
        3. $\pi[v] = ce$; $pred[v] = e$;

Demo: Prim's algorithm

Prim's algorithm: analysis

Theorem. Prim's algorithm can be implemented to run in $O(m \log n)$ time.
Pf. Implementation almost identical to Dijkstra's algorithm.

Depends on `PQ`

- n `INSERT`,
- n `DELETE-MIN`,
- $\leq m$ `DECREASE-KEY`.

Kruskal's algorithm

Consider edges in *ascending* order of cost:

- Add to tree unless it would create a cycle.

Kruskal's algorithm

Consider edges in *ascending* order of cost:

- Add to tree unless it would create a cycle.

Theorem. Kruskal's algorithm computes an MST.
Pf. Special case of greedy coloring.

- Case 1: both endpoints of e in same blue tree.
  - color e red by applying red rule to unique cycle.
- Case 2: endpoints of e in different blue trees.
  - color e blue by applying blue rule to cutset defined by either tree.

Union-Find data structure

Pointer-based implementation

- `MAKE-SET`(v): $O(n)$
- `FIND-SET`(v): $O(\log n)$
- `UNION`(u, v): 1

Kruskal's algorithm: implementation

1. SORT m edges by cost and renumber so that $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$;
2. $T = \emptyset$;
3. FOREACH $v \in V$: `MAKE-SET`(v);
4. FOR $i = 1..m$:
   1. $(u, v) = e_i$;
   2. IF (`FIND-SET`(u) $\neq$ `FIND-SET`(v)):
      1. $T = T \cup e_i$;
      2. `UNION`(u, v);
5. RETURN T.

Demo: Kruskal's algorithm

Kruskal's algorithm: analysis

Theorem. Kruskal's algorithm can be implemented to run in $O(m \log m)$ time.

Pf.

- Sort edges by cost.
- Use `union-find` data structure to dynamically maintain connected components.

Reverse-delete algorithm

Consider edges in *descending* order of cost:

- Start with all edges in T.
- Delete edge from T unless it would disconnect T.

Reverse-delete algorithm

Consider edges in *descending* order of cost:

- Start with all edges in T.
- Delete edge from T unless it would disconnect T.

Theorem. The reverse-delete algorithm computes an MST.
Pf. Special case of greedy coloring.

- Case 1. [deleting edge e does not disconnect T]
  - apply red rule to cycle C formed by adding e to another path in T between its two endpoints
- Case 2. [deleting edge e disconnects T]
  - apply blue rule to cutset D induced by either component

Reverse-delete algorithm

Consider edges in *descending* order of cost:

- Start with all edges in T.
- Delete edge from T unless it would disconnect T.

Theorem. The reverse-delete algorithm computes an MST.
Pf. Special case of greedy coloring.

- Case 1. [deleting edge e does not disconnect T]
  - apply red rule to cycle C formed by adding e to another path in T between its two endpoints
- Case 2. [deleting edge e disconnects T]
  - 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.

Demo: Reverse-delete algorithm

Review: greedy MST algorithms

Red rule.

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

Blue rule.

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

Greedy coloring.

- 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.

Review: greedy MST algorithms

Red rule.

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

Blue rule.

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

Greedy coloring.

- 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.

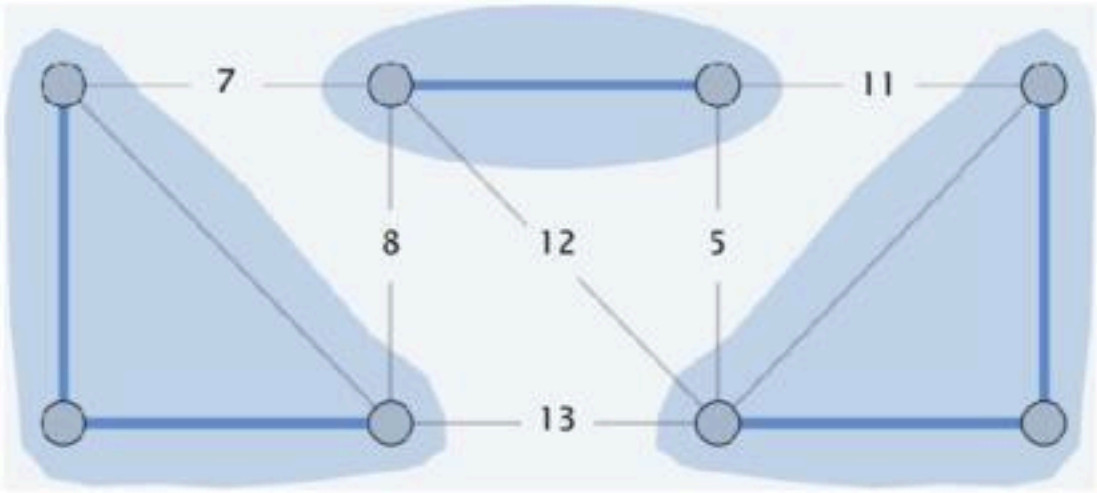
- Apply blue rule to cutset corresponding to *each* blue tree.
- Color *all* selected edges blue.

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 coloring (repeatedly apply blue rule).



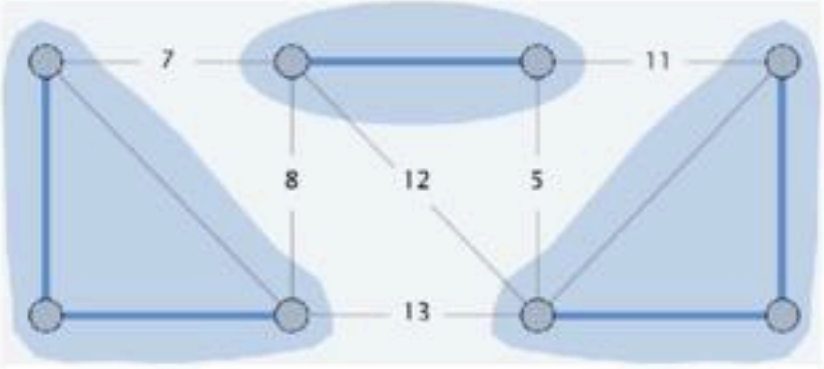
Demo: Borůvka's algorithm

Borůvka's: analysis

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

Pf.

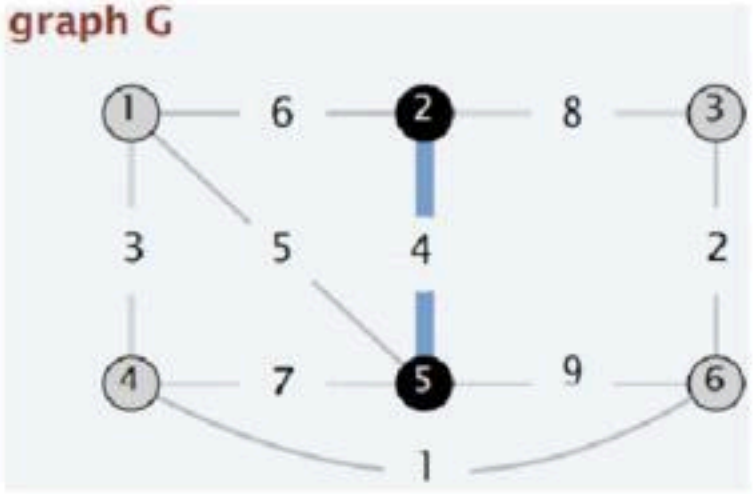
- To implement a phase in $O(m)$ time:
  - compute connected components of blue edges: $O(m)$
  - for each edge $(u, v) \in E$, check if u and v are in different components: $O(\log m)$;
    - if so, update each component's best edge in cutset
- $\leq \log_2 n$ phases since each phase (at least) halves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

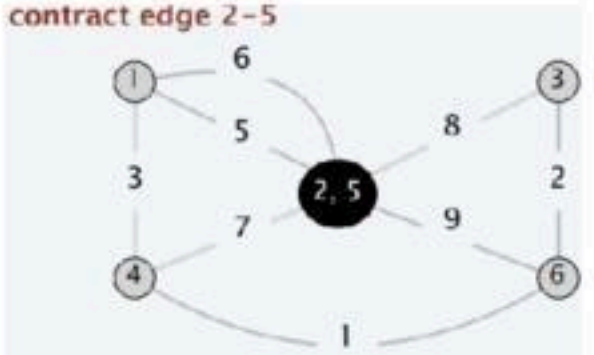


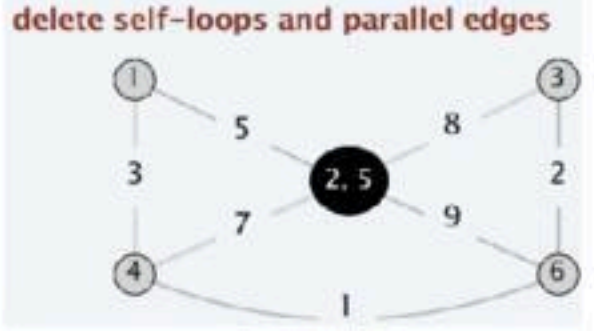
Borůvka's: Contraction implementation

Contraction version.

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



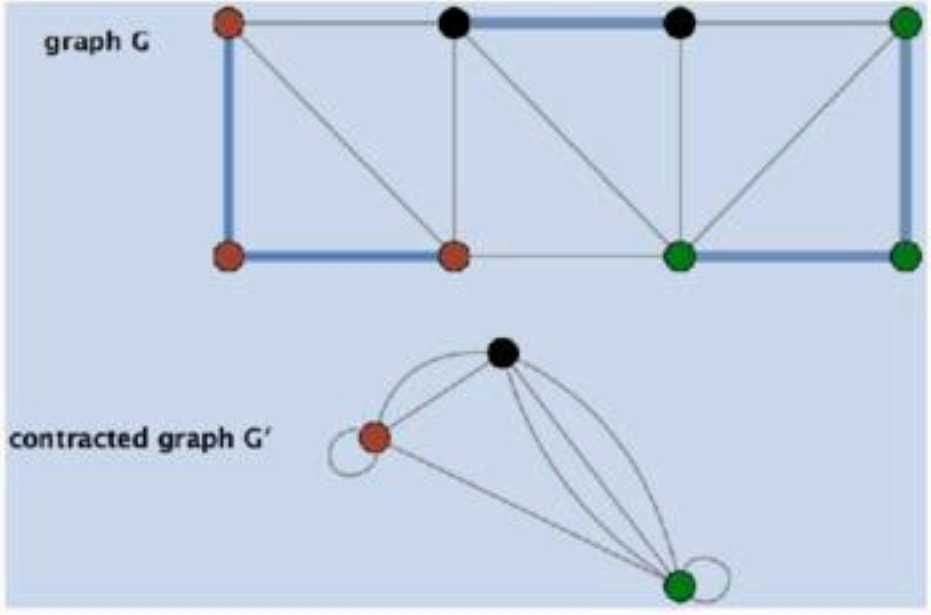




Contract a set of edges

Problem. Given a graph $G = (V, E)$ and a set of edges F, contract all edges in F, removing any self-loops or parallel edges.

Goal. $O(m + n)$ time.



Contract a set of edges: solution

Problem. Given a graph $G = (V, E)$ and a set of edges F, contract all edges in F, removing any self-loops or parallel edges.

Solution.

- Compute the n' connected components in (V, F).
- Suppose $id[u] = i$ means node u is in connected component i.
- The contracted graph G' has n' nodes.
- For each edge u–$v \in E$, add an edge i–j to G', where $i = id[u]$ and $j = id[v]$.

Removing self loops: Easy.

Removing parallel edges.

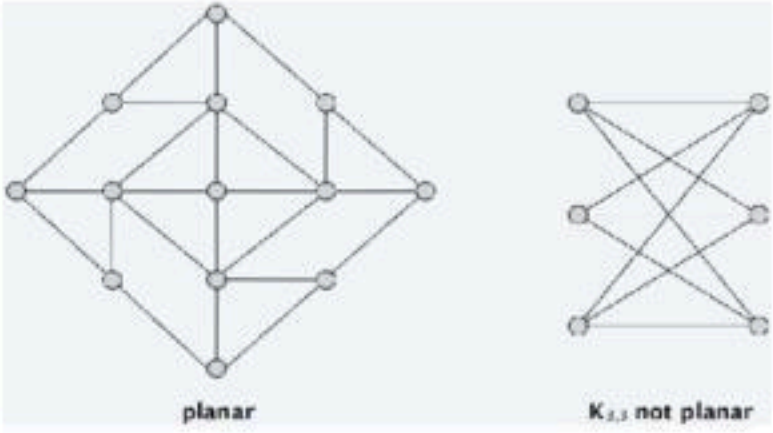
- Create a list of edges i–j with the convention that $i < j$.
- Sort the edges lexicographically via `LSD radix sort`.
- Add the edges to the graph G', removing parallel edges.

Borůvka's algorithm on planar graphs

Theorem. Borůvka's algorithm (contraction version) can be implemented to run in $O(n)$ time on planar graphs.

Pf.

- Each Borůvka phase takes $O(n)$ time:
  - Fact 1: $m \leq 3n$ for simple planar graphs.
  - Fact 2: planar graphs remains planar after edge contractions/deletions.
- Number of nodes (at least) halves in each phase.
- Thus, overall running time $\leq cn + cn/2 + cn/4 + cn/8 + \ldots = O(n)$.



Borůvka–Prim algorithm

Borůvka–Prim algorithm.

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

Theorem. Borůvka–Prim computes an MST.
Pf. Special case of the greedy algorithm.

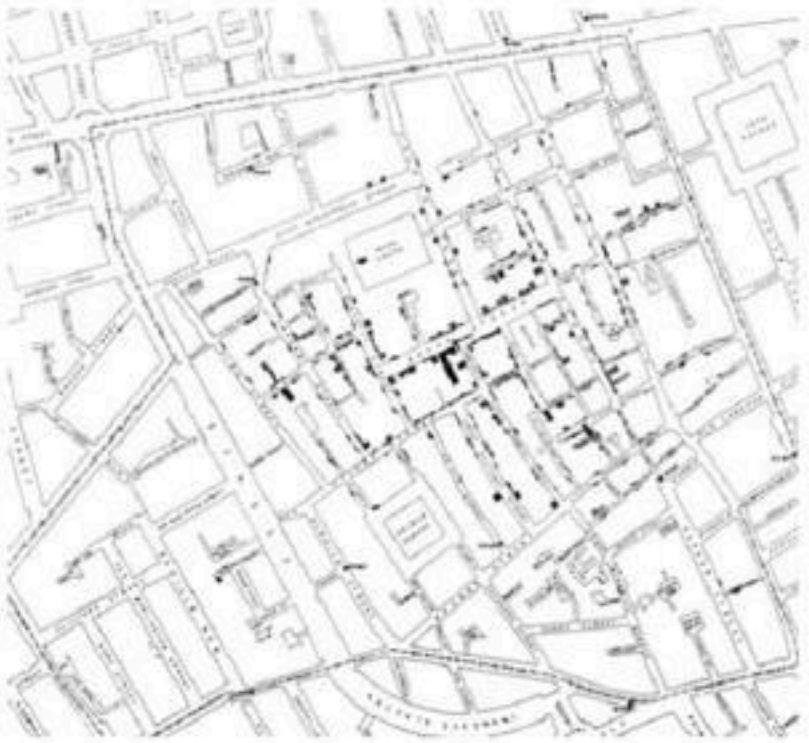
Theorem. Borůvka–Prim can be implemented to run in $O(m \log \log n)$ time.
Pf.

- The $\log_2 \log_2 n$ phases of Borůvka's algorithm take $O(m \log \log n)$ time;
  - resulting graph has $\leq n / \log_2 n$ nodes and $\leq m$ edges.
- Prim's algorithm (using Fibonacci heaps) takes $O(m + n)$ time on a graph with $n / \log_2 n$ nodes and m edges.
  - precisely, $O(m + \frac{n}{\log n} \log(\frac{n}{\log n}))$.

Single-link clustering

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.



Clustering of maximum spacing

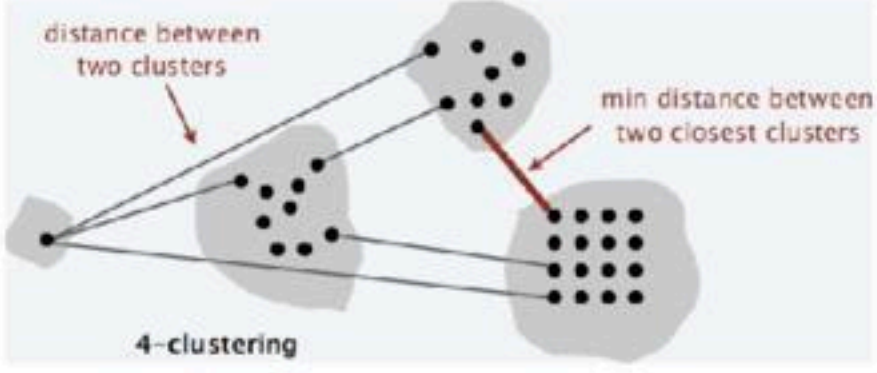
k-**clustering**. Divide objects into k non-empty groups.

Distance function. Numeric value specifying "closeness" of two objects.

- $d(p_i, p_j) = 0$ iff $p_i = p_j$ [identity]
- $d(p_i, p_j) \geq 0$ [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).

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.

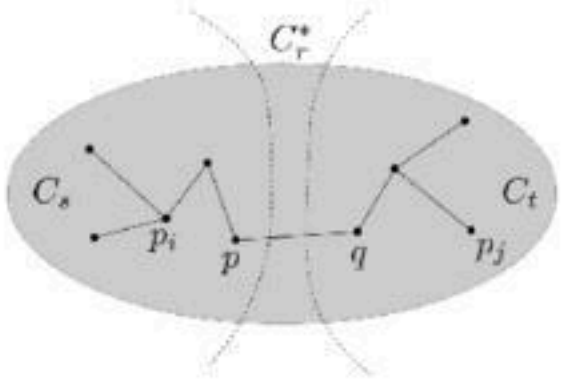
Greedy clustering: analysis

Theorem. Let $C^* = C_1^*, \ldots, C_k^*$ denote the clustering formed by deleting k–1 longest edges of an MST. Then, C^* is a k-clustering of max spacing.

Pf.

Spacing of C^* = length d^* of the $(k$–$1)^{st}$ longest edge in MST.

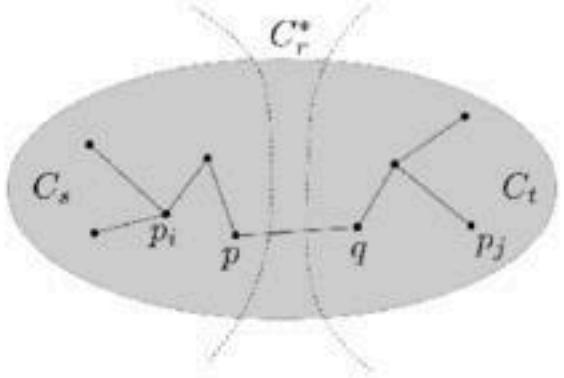
- Consider any other clustering $C = C_1, \ldots, C_k$.
  - need to show C has a smaller spacing



Greedy clustering: analysis (cont.)

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_r^* spans two different clusters in C.
- Edge (p, q) has length $\leq d^*$ since it was added by Kruskal.
- Spacing of C is $\leq d^*$ since p and q are in different clusters.

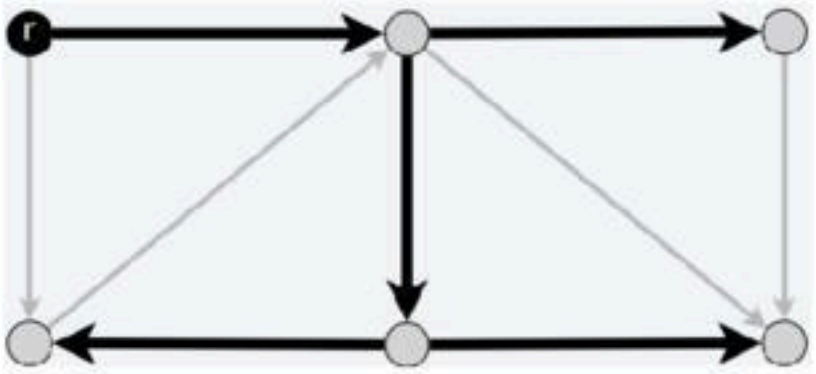


Min-cost arborescence

Arborescence

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

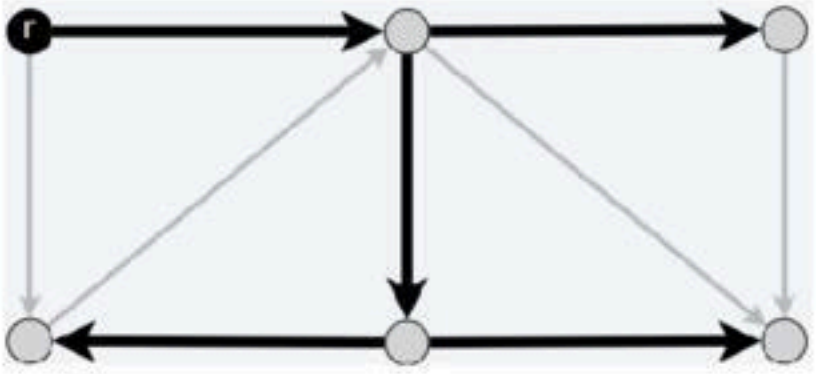
- T is a spanning tree of G if we ignore the direction of edges.
- There is a (unique) directed path in T from r to each other node $v \in V$.



Quiz: Arborescence

Which of the following are properties of arborescence rooted at r?

A. No directed cycles.
B. Exactly $n - 1$ edges.
C. For each $v \neq r$: $indegree(v) = 1$.
D. All of the above.



Arborescence: property

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.

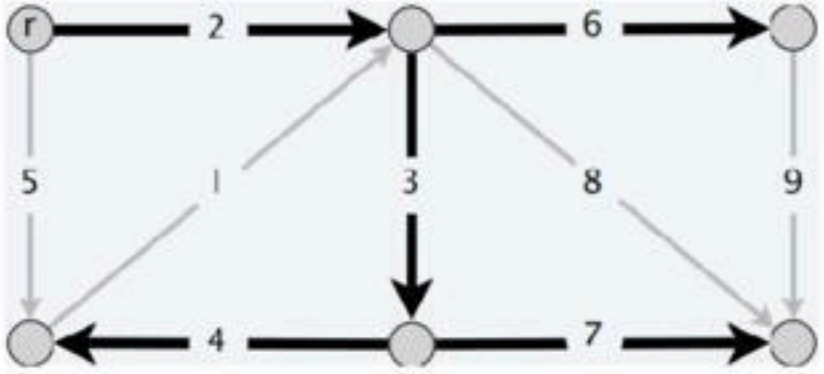
$\Rightarrow$ If T is an arborescence, then no (directed) cycles and every node $v \neq r$ has exactly one entering edge: the last edge on the unique $r \rightsquigarrow v$ path.

$\Leftarrow$ Suppose T has no cycles and each node $v \neq r$ has one entering edge.

- To construct an $r \rightsquigarrow v$ 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 edge costs $c_e \geq 0$, find an arborescence rooted at r of minimum cost.



Assumption 1. All nodes reachable from r.

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

Quiz: Minimum spanning arborescence

A min-cost arborescence must ...

A. Include the cheapest edge.
B. Exclude the most expensive edge.
C. Be a shortest-paths tree from r.
D. None of the above.

Quiz: Minimum spanning arborescence

A min-cost arborescence must ...

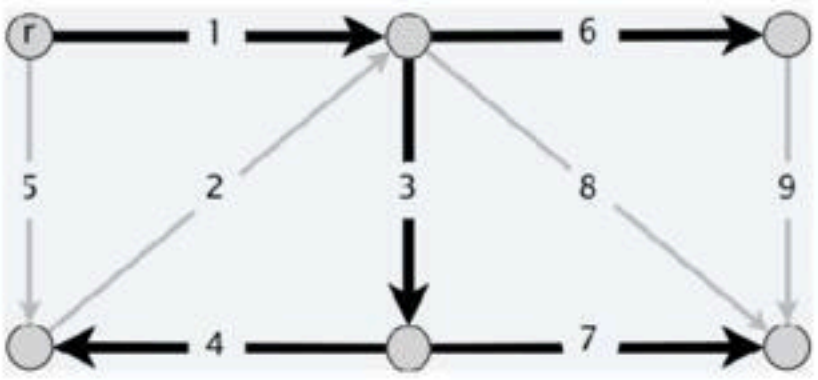
A. Include the cheapest edge.
B. Exclude the most expensive edge.
C. Be a shortest-paths tree from r.
D. None of the above.

D. See below.

A sufficient optimality condition

Property. For each node $v \neq r$, choose a 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.

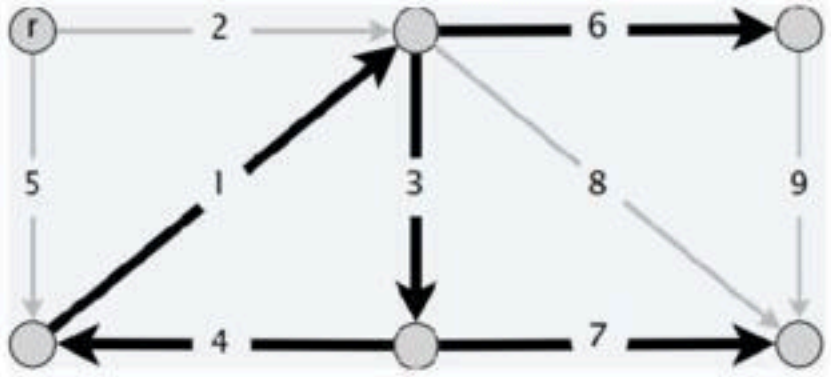
Pf. An arborescence needs exactly one edge entering each node $v \neq r$ and (V, F^*) is the cheapest way to make each of these choices.



A sufficient optimality condition

Property. For each node $v \neq r$, choose a 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.

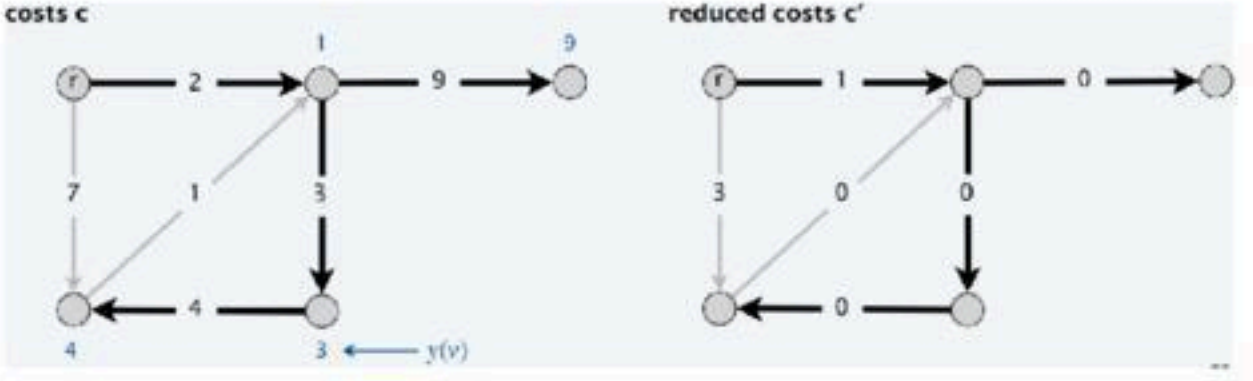
Note. F^* may not be an arborescence (since it may have directed cycles).



Reduced costs

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

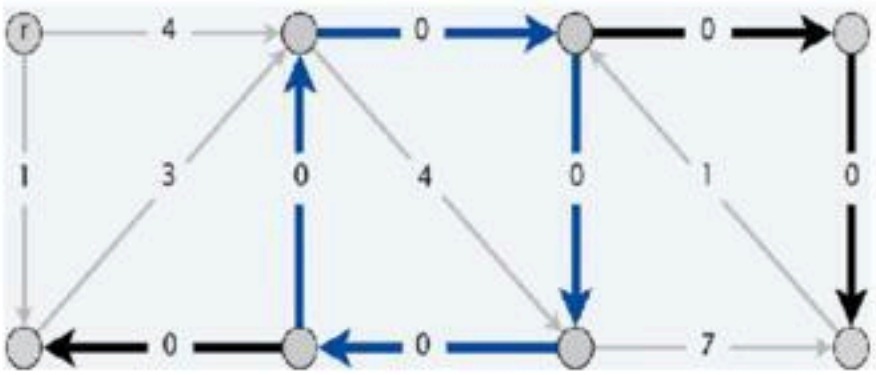
Observation. T is a min-cost arborescence in G using costs c iff T is a min-cost arborescence in G using reduced costs c'.
Pf. For each $v \neq r$: each arborescence has exactly one edge entering v.

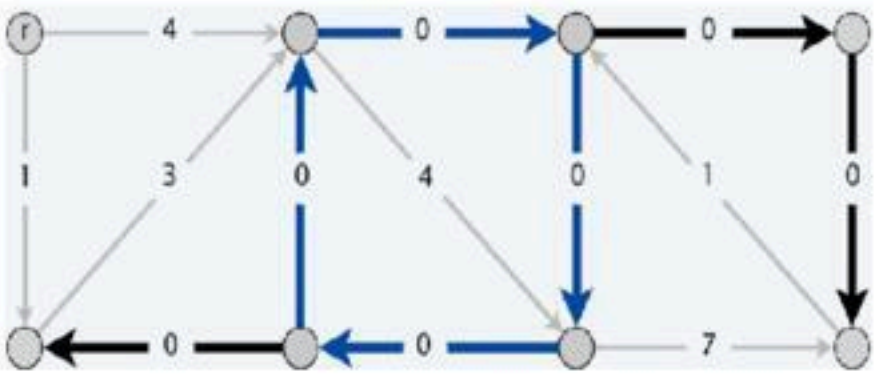


Edmonds branching algorithm: intuition

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

- Now, all edges in F^* have 0 cost with respect to reduced 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* edges in C to a supernode (removing any self-loops).
    - Recursively solve problem in contracted network G' with costs $c'(u, v)$.





Edmonds branching algorithm

1. FOREACH $v \neq r$:
   1. $y(v)$ = min cost of any edge entering v;
   2. $c'(u,v) = c'(u,v) - y(v)$ for each edge (u,v) entering v;
2. FOREACH $v \neq r$: choose one 0-cost edge entering v and let F^* be the resulting set of edges;
3. IF (F^* forms an arborescence): RETURN $T = (V, F^*)$;
4. ELSE:
   1. C = directed cycle in F^*;
   2. Contract C to a single supernode, yielding $G' = (V', E')$;
   3. T' = EDMONDS-BRANCHING(G', r, c');
   4. Extend T' to an arborescence T in G by adding all but one edge of C;
   5. RETURN T;

Demo: Edmonds branching algorithm

Edmonds branching algorithm: all done?

Q. What could go wrong?

A. Contracting cycle C places extra constraint on arborescence.

- Min-cost arborescence in G' must have exactly one edge entering a node in C (since C is contracted to a single node)
  - But min-cost arborescence in G might have several edges entering C.

Edmonds branching: key lemma

Lemma. Let C be a cycle in G containing only 0-cost edges. There exists a min-cost arborescence T rooted at r that has exactly one edge entering C.
Pf.

Case 0. T has no edges entering C.

- Since T is an arborescence, there is an $r \rightsquigarrow v$ path for each node v
  - at least one edge enters C.

Case 1. T has exactly one edge entering C.

- T satisfies the lemma, done.

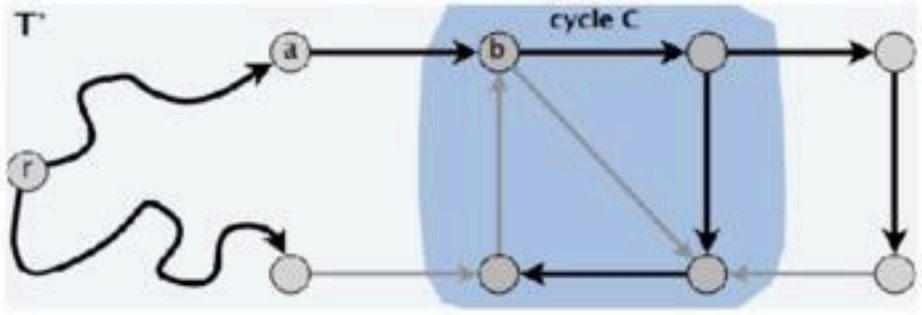
Case 2. T has two (or more) edges entering C.

- We construct another min-cost arborescence T^* that has exactly one edge entering C.

Edmonds branching: key lemma (cont.)

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: analysis

Theorem. [Chu–Liu 1965, Edmonds 1967] The greedy algorithm finds a min-cost arborescence.

Pf. [by strong induction on number of nodes]

- 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.

Edmonds branching: better bound

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