CSCI 5451 Fall 2015

Week 9 Notes

Professor Ellen Gethner

October 14, 2015

Shortest Path Problems, continued from last week

How to Grow a Tree

- **Definition.** Let G be a simple graph and T a subgraph of G that is a tree.

Shortest Path Problems, continued from last week

How to Grow a Tree

- **Definition.** Let G be a simple graph and T a subgraph of G that is a tree.

- A **frontier edge** of T is an edge $uv \in E(G)$ such that $u \in V(T)$ and $v \notin V(T)$.

Shortest Path Problems, continued from last week

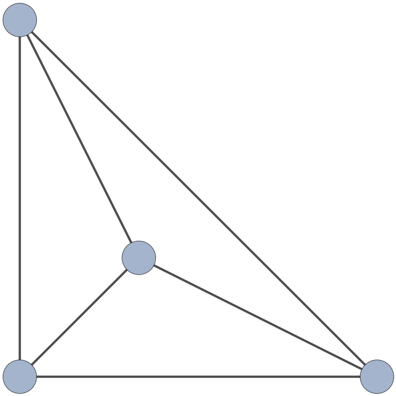
How to Grow a Tree

- **Definition.** Let G be a simple graph and T a subgraph of G that is a tree.

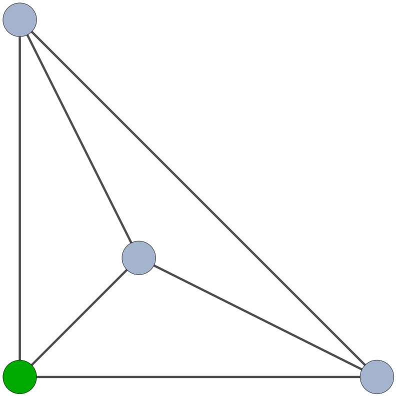
- A **frontier edge** of T is an edge $uv \in E(G)$ such that $u \in V(T)$ and $v \notin V(T)$.

- **Example.** $G = K_4$

Example: frontier edges

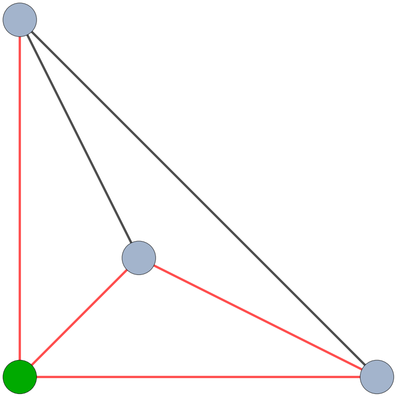


Example: frontier edges



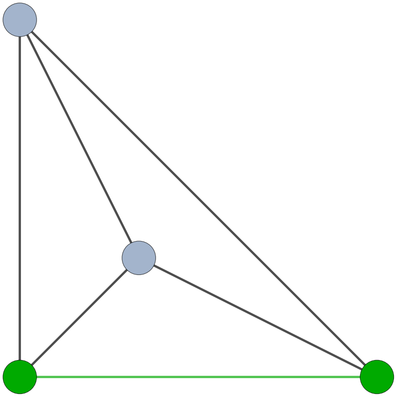
Tree T, so far, is the single green vertex.

Example: frontier edges



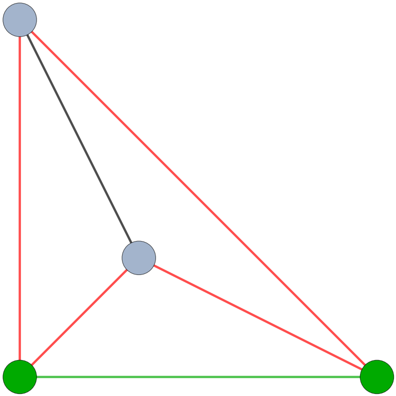
The current frontier edges are red.

Example: frontier edges



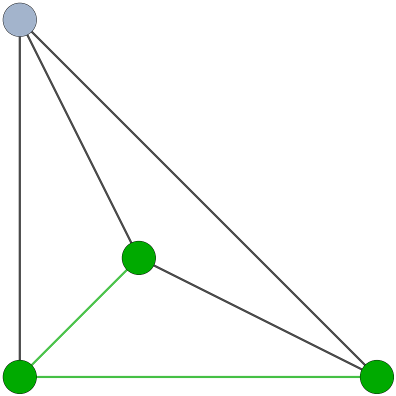
Choose a frontier edge and update the tree.

Example: frontier edges



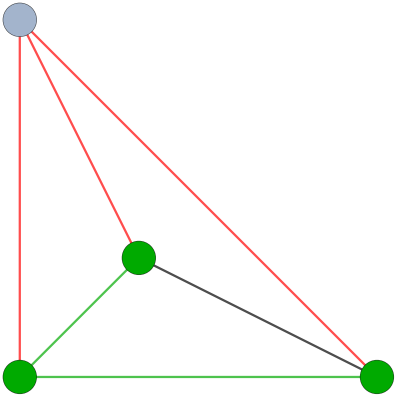
Now update the frontier edges.

Example: frontier edges



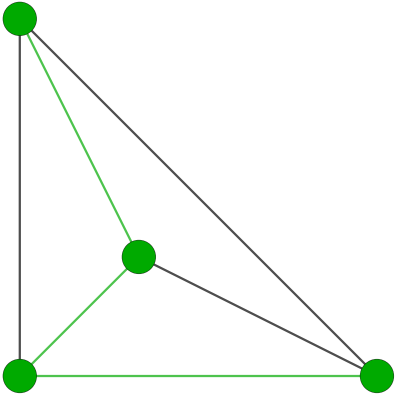
Choose any frontier edge and update the tree.

Example: frontier edges



And update the frontier edges again.

Example: frontier edges



And finally, choose one last frontier edge and update the tree.
Why are we done?

Tree Growing Algorithm

- **Input:** A connected graph G and starting vertex $v \in V(G)$
- **Output:** A spanning tree T of G and a vertex labeling of $V(G)$.
- **initialize** T as vertex v
- **initialize** set of frontier edges for T as **empty**
- **write** label 0 on vertex v
- **initialize** label counter $i = 1$
- **while** T does not yet span G
- **update** the set of frontier edges for T
- **let** e be the frontier edge for T of highest priority
- **let** w be the unlabeled endpoint of e
- **add** edge e (and vertex w) to tree T
- **write** label i on vertex w
- $i = i + 1$
- **return** T with its vertex labeling

An application of tree growing

- Back at the ranch we were interested in finding the shortest path from some root to each other vertex in a graph.

An application of tree growing

- Back at the ranch we were interested in finding the shortest path from some root to each other vertex in a graph.

- Two things that are missing so far from our tree growing algorithm are

An application of tree growing

- Back at the ranch we were interested in finding the shortest path from some root to each other vertex in a graph.

- Two things that are missing so far from our tree growing algorithm are

  1. the fact that the edges are weighted, and

An application of tree growing

- Back at the ranch we were interested in finding the shortest path from some root to each other vertex in a graph.

- Two things that are missing so far from our tree growing algorithm are

  1. the fact that the edges are weighted, and
  2. a way to assign priorities to the frontier edges.

An application of tree growing

- Back at the ranch we were interested in finding the shortest path from some root to each other vertex in a graph.

- Two things that are missing so far from our tree growing algorithm are

  1. the fact that the edges are weighted, and

  2. a way to assign priorities to the frontier edges.

- So next let's use the weights of the edges to prioritize frontier edges in a way that makes sense for an eventual shortest path algorithm.

Prioritizing Frontier Edges

- Let G be a simple undirected weighted graph and suppose T is a subgraph of G that is a tree.

Prioritizing Frontier Edges

- Let G be a simple undirected weighted graph and suppose T is a subgraph of G that is a tree.

- For each frontier edge e of T, define the **priority of** e by $P(e) = d(s, x) + \omega(e)$, where x is the labeled endpoint of e;

Prioritizing Frontier Edges

- Let G be a simple undirected weighted graph and suppose T is a subgraph of G that is a tree.

- For each frontier edge e of T, define the **priority of** e by $P(e) = d(s, x) + \omega(e)$, where x is the labeled endpoint of e;

- recall that $d(s, x)$ is the distance from vertex s to vertex x, and $\omega(e)$ is the weight of edge e (see week 8's notes).

Prioritizing Frontier Edges

- Let G be a simple undirected weighted graph and suppose T is a subgraph of G that is a tree.
- For each frontier edge e of T, define the **priority of** e by $P(e) = d(s,x) + \omega(e)$, where x is the labeled endpoint of e;
- recall that $d(s,x)$ is the distance from vertex s to vertex x, and $\omega(e)$ is the weight of edge e (see week 8's notes).
- Now that we know how to prioritize the frontier edges, we have all of the tools needed to describe Dijkstra's Single Source Shortest Path Algorithm.

Algorithm Dijkstra(G, s)

- **Input.** A weighted, connected, undirected graph G whose edge weights are non-negative; a source vertex $s \in V(G)$.

Algorithm Dijkstra(G, s)

- **Input.** A weighted, connected, undirected graph G whose edge weights are non-negative; a source vertex $s \in V(G)$.

- **Output.** A spanning tree T (the "Dijkstra Tree") of G, rooted at vertex s such that

Algorithm Dijkstra(G, s)

- **Input.** A weighted, connected, undirected graph G whose edge weights are non-negative; a source vertex $s \in V(G)$.

- **Output.** A spanning tree T (the "Dijkstra Tree") of G, rooted at vertex s such that

  1. the unique path from s to each vertex $v \in V(G)$ in T is a shortest path from s to v in G, and

Algorithm Dijkstra(G, s)

- **Input.** A weighted, connected, undirected graph G whose edge weights are non-negative; a source vertex $s \in V(G)$.
- **Output.** A spanning tree T (the "Dijkstra Tree") of G, rooted at vertex s such that

  1. the unique path from s to each vertex $v \in V(G)$ in T is a shortest path from s to v in G, and
  2. a vertex labeling giving the distance from s to each vertex.

Algorithm Dijkstra(G, s), continued

- **initialize** the Dijkstra tree T as the given source vertex s

Algorithm Dijkstra(G, s), continued

- **initialize** the Dijkstra tree T as the given source vertex s
- **initialize** the set of frontier edges as **empty**

Algorithm Dijkstra(G, s), continued

- **initialize** the Dijkstra tree T as the given source vertex s
- **initialize** the set of frontier edges as **empty**
- $dist[s] = 0$

Algorithm Dijkstra(G, s), continued

- **initialize** the Dijkstra tree T as the given source vertex s
- **initialize** the set of frontier edges as **empty**
- $dist[s] = 0$
- **write** label 0 on vertex s

Algorithm Dijkstra(G, s), continued

- **initialize** the Dijkstra tree T as the given source vertex s
- **initialize** the set of frontier edges as **empty**
- $dist[s] = 0$
- **write** label 0 on vertex s
- **while** the Djikstra tree T does not yet span G

Algorithm Dijkstra(G, s), continued

- **initialize** the Dijkstra tree T as the given source vertex s
- **initialize** the set of frontier edges as **empty**
- $dist[s] = 0$
- **write** label 0 on vertex s
- **while** the Djikstra tree T does not yet span G
-     **update** the frontier edges for T

Algorithm Dijkstra(G, s), continued

- **initialize** the Dijkstra tree T as the given source vertex s
- **initialize** the set of frontier edges as **empty**
- $dist[s] = 0$
- **write** label 0 on vertex s
- **while** the Djikstra tree T does not yet span G
-     **update** the frontier edges for T
-     **for** each frontier edge $e = xy$ for T

Algorithm Dijkstra(G, s), continued

- **initialize** the Dijkstra tree T as the given source vertex s
- **initialize** the set of frontier edges as **empty**
- $dist[s] = 0$
- **write** label 0 on vertex s
- **while** the Djikstra tree T does not yet span G
-     **update** the frontier edges for T
-     **for** each frontier edge $e = xy$ for T
-         **let** x be the labeled endpoint of e

Algorithm Dijkstra(G, s), continued

- **initialize** the Dijkstra tree T as the given source vertex s
- **initialize** the set of frontier edges as **empty**
- $dist[s] = 0$
- **write** label 0 on vertex s
- **while** the Djikstra tree T does not yet span G
-     **update** the frontier edges for T
-     **for** each frontier edge $e = xy$ for T
-         **let** x be the labeled endpoint of e
-         **let** y be the unlabeled endpoint of e

Algorithm Dijkstra(G, s), continued

- **initialize** the Dijkstra tree T as the given source vertex s
- **initialize** the set of frontier edges as **empty**
- $dist[s] = 0$
- **write** label 0 on vertex s
- **while** the Djikstra tree T does not yet span G
-     **update** the frontier edges for T
-     **for** each frontier edge $e = xy$ for T
-         **let** x be the labeled endpoint of e
-         **let** y be the unlabeled endpoint of e
-         **set** $P(e) = dist[x] + \omega(e)$.

Algorithm Dijkstra(G, s), continued

- **initialize** the Dijkstra tree T as the given source vertex s
- **initialize** the set of frontier edges as **empty**
- $dist[s] = 0$
- **write** label 0 on vertex s
- **while** the Djikstra tree T does not yet span G
-     **update** the frontier edges for T
-     **for** each frontier edge $e = xy$ for T
-         **let** x be the labeled endpoint of e
-         **let** y be the unlabeled endpoint of e
-         **set** $P(e) = dist[x] + \omega(e)$.
-     **let** $e = xy$ be a frontier edge of T for which $P(e)$ is smallest

Algorithm Dijkstra(G, s), continued

- **initialize** the Dijkstra tree T as the given source vertex s
- **initialize** the set of frontier edges as **empty**
- $dist[s] = 0$
- **write** label 0 on vertex s
- **while** the Djikstra tree T does not yet span G
-     **update** the frontier edges for T
-     **for** each frontier edge $e = xy$ for T
-         **let** x be the labeled endpoint of e
-         **let** y be the unlabeled endpoint of e
-         **set** $P(e) = dist[x] + \omega(e)$.
-     **let** $e = xy$ be a frontier edge of T for which $P(e)$ is smallest
-     **let** x be the unlabeled endpoint of e and y be the labeled endpoint of e

Algorithm Dijkstra(G, s), continued

- **initialize** the Dijkstra tree T as the given source vertex s
- **initialize** the set of frontier edges as **empty**
- $dist[s] = 0$
- **write** label 0 on vertex s
- **while** the Djikstra tree T does not yet span G
- **update** the frontier edges for T
- **for** each frontier edge $e = xy$ for T
- **let** x be the labeled endpoint of e
- **let** y be the unlabeled endpoint of e
- **set** $P(e) = dist[x] + \omega(e)$.
- **let** $e = xy$ be a frontier edge of T for which $P(e)$ is smallest
- **let** x be the unlabeled endpoint of e and y be the labeled endpoint of e
- **add** edge e (and vertex y) to tree T

Algorithm Dijkstra(G, s), continued

- **initialize** the Dijkstra tree T as the given source vertex s
- **initialize** the set of frontier edges as **empty**
- $dist[s] = 0$
- **write** label 0 on vertex s
- **while** the Djikstra tree T does not yet span G
-     **update** the frontier edges for T
-     **for** each frontier edge $e = xy$ for T
-         **let** x be the labeled endpoint of e
-         **let** y be the unlabeled endpoint of e
-         **set** $P(e) = dist[x] + \omega(e)$.
-     **let** $e = xy$ be a frontier edge of T for which $P(e)$ is smallest
-     **let** x be the unlabeled endpoint of e and y be the labeled endpoint of e
-     **add** edge e (and vertex y) to tree T
-     $dist[y] = P(e)$

Algorithm Dijkstra(G, s), continued

- **initialize** the Dijkstra tree T as the given source vertex s
- **initialize** the set of frontier edges as **empty**
- $dist[s] = 0$
- **write** label 0 on vertex s
- **while** the Djikstra tree T does not yet span G
- **update** the frontier edges for T
- **for** each frontier edge $e = xy$ for T
- **let** x be the labeled endpoint of e
- **let** y be the unlabeled endpoint of e
- **set** $P(e) = dist[x] + \omega(e)$.
- **let** $e = xy$ be a frontier edge of T for which $P(e)$ is smallest
- **let** x be the unlabeled endpoint of e and y be the labeled endpoint of e
- **add** edge e (and vertex y) to tree T
- $dist[y] = P(e)$
- **write** label $dist(y)$ on vertex y.

Algorithm Dijkstra(G, s), continued

- **initialize** the Dijkstra tree T as the given source vertex s
- **initialize** the set of frontier edges as **empty**
- $dist[s] = 0$
- **write** label 0 on vertex s
- **while** the Djikstra tree T does not yet span G
  - **update** the frontier edges for T
  - **for** each frontier edge $e = xy$ for T
    - **let** x be the labeled endpoint of e
    - **let** y be the unlabeled endpoint of e
    - **set** $P(e) = dist[x] + \omega(e)$.
  - **let** $e = xy$ be a frontier edge of T for which $P(e)$ is smallest
  - **let** x be the unlabeled endpoint of e and y be the labeled endpoint of e
  - **add** edge e (and vertex y) to tree T
  - $dist[y] = P(e)$
  - **write** label $dist(y)$ on vertex y.
- **return** Dijkstra tree T and its vertex labels.

Thoughts about the algorithm

- **Remark.** Each time a new vertex y (together with its associated edge) is added to the Djikstra tree, no new computations are needed to update the frontier edges;

Thoughts about the algorithm

- **Remark.** Each time a new vertex y (together with its associated edge) is added to the Djikstra tree, no new computations are needed to update the frontier edges;
- simply delete frontier edges that include the endpoint y.

Thoughts about the algorithm

- **Remark.** Each time a new vertex y (together with its associated edge) is added to the Djikstra tree, no new computations are needed to update the frontier edges;
- simply delete frontier edges that include the endpoint y.
- Then recompute the priorities of the remaining edges.

Thoughts about the algorithm

- **Remark.** Each time a new vertex y (together with its associated edge) is added to the Djikstra tree, no new computations are needed to update the frontier edges;
- simply delete frontier edges that include the endpoint y.
- Then recompute the priorities of the remaining edges.
- **Question.** Is Djikstra's algorithm correct?

Proof of Correctness of Djikstra's Algorithm

- **Theorem.** Let T_j be the Dijkstra tree after j iterations of Algorithm Dijkstra on connect graph G for $j = 0, 1, 2, \ldots, |V(G)| - 1$.Then for each $v \in V(T_j)$, the unique path from s to v in T_j is a shortest path from s to v in G.

Proof of Correctness of Djikstra's Algorithm

- **Theorem.** Let T_j be the Dijkstra tree after j iterations of Algorithm Dijkstra on connect graph G for $j = 0, 1, 2, \ldots, |V(G)| - 1$.Then for each $v \in V(T_j)$, the unique path from s to v in T_j is a shortest path from s to v in G.

- **Proof by induction.**

Proof of Correctness of Djikstra's Algorithm

- **Theorem.** Let T_j be the Dijkstra tree after j iterations of Algorithm Dijkstra on connect graph G for $j = 0, 1, 2, \ldots, |V(G)| - 1$.Then for each $v \in V(T_j)$, the unique path from s to v in T_j is a shortest path from s to v in G.

- **Proof by induction.**
- **Base Cases.**

Proof of Correctness of Djikstra's Algorithm

- **Theorem.** Let T_j be the Dijkstra tree after j iterations of Algorithm Dijkstra on connect graph G for $j = 0, 1, 2, \ldots, |V(G)| - 1$.Then for each $v \in V(T_j)$, the unique path from s to v in T_j is a shortest path from s to v in G.

- **Proof by induction.**
- **Base Cases.**
  - The claim is true for T_0. Why?

Proof of Correctness of Djikstra's Algorithm

- **Theorem.** Let T_j be the Dijkstra tree after j iterations of Algorithm Dijkstra on connect graph G for $j = 0, 1, 2, \ldots, |V(G)| - 1$.Then for each $v \in V(T_j)$, the unique path from s to v in T_j is a shortest path from s to v in G.

- **Proof by induction.**
- **Base Cases.**
  - The claim is true for T_0. Why?
  - What must you check to know that the claim is true for T_1?

Proof of correctness of Dijkstra, continued

- **Induction Hypothesis.** Assume for some fixed j such that $0 \leq j \leq |V(G)| - 1$ that T_j satisfies the claim.

Proof of correctness of Dijkstra, continued

- **Induction Hypothesis.** Assume for some fixed j such that $0 \leq j \leq |V(G)| - 1$ that T_j satisfies the claim.

- **Inductive Step.** Let $e \in E(G)$ with $e = xy$ and suppose $x \in V(T_j)$ and $y \notin V(T_j)$.

Proof of correctness of Dijkstra, continued

- **Induction Hypothesis.** Assume for some fixed j such that $0 \leq j \leq |V(G)| - 1$ that T_j satisfies the claim.
- **Inductive Step.** Let $e \in E(G)$ with $e = xy$ and suppose $x \in V(T_j)$ and $y \notin V(T_j)$.
- In particular, e is a frontier edge, x is labeled, and y is unlabeled.

Proof of correctness of Dijkstra, continued

- **Induction Hypothesis.** Assume for some fixed j such that $0 \leq j \leq |V(G)| - 1$ that T_j satisfies the claim.

- **Inductive Step.** Let $e \in E(G)$ with $e = xy$ and suppose $x \in V(T_j)$ and $y \notin V(T_j)$.

- In particular, e is a frontier edge, x is labeled, and y is unlabeled.

- Suppose e is the frontier edge added to T_j in the $(j+1)$st iteration of Algorithm Dijkstra.

Proof of correctness of Dijkstra, continued

- **Induction Hypothesis.** Assume for some fixed j such that $0 \leq j \leq |V(G)| - 1$ that T_j satisfies the claim.

- **Inductive Step.** Let $e \in E(G)$ with $e = xy$ and suppose $x \in V(T_j)$ and $y \notin V(T_j)$.

- In particular, e is a frontier edge, x is labeled, and y is unlabeled.

- Suppose e is the frontier edge added to T_j in the $(j+1)$st iteration of Algorithm Dijkstra.

- Since y is the only new vertex in T_{j+1}, it suffices to show that the path from s to y in T_{j+1} is a shortest path from s to y in G.

Proof of correctness of Dijkstra, continued

- Let Q be the (unique) path from s to y in T_{j+1}.

Proof of correctness of Dijkstra, continued

- Let Q be the (unique) path from s to y in T_{j+1}.
- **Observation.** The path Q contains a path from s to x in T_j and adds vertex y (and hence edge e) to form a path from s to y.

Proof of correctness of Dijkstra, continued

- Let Q be the (unique) path from s to y in T_{j+1}.
- **Observation.** The path Q contains a path from s to x in T_j and adds vertex y (and hence edge e) to form a path from s to y.
- Thus $\mathit{length}(Q) = P(e)$. Why?

Proof of correctness of Dijkstra, continued

- Let Q be the (unique) path from s to y in T_{j+1}.
- **Observation.** The path Q contains a path from s to x in T_j and adds vertex y (and hence edge e) to form a path from s to y.
- Thus $\mathit{length}(Q) = P(e)$. Why?
- Now let R be *any* path in G from s to y.

Proof of correctness of Dijkstra, continued

- Let Q be the (unique) path from s to y in T_{j+1}.
- **Observation.** The path Q contains a path from s to x in T_j and adds vertex y (and hence edge e) to form a path from s to y.
- Thus $length(Q) = P(e)$. Why?
- Now let R be *any* path in G from s to y.
- To finish the proof, it suffices to show that $length(R) \geq length(Q)$. Why?

Proof of correctness of Dijkstra, continued

- WLOG assume R and Q are different and let edge $f = vz$ be the first edge in path R that is not in T_j.

Proof of correctness of Dijkstra, continued

- WLOG assume R and Q are different and let edge $f = vz$ be the first edge in path R that is not in T_j.
- Assume also that $v \in V(T_j)$ (and hence $z \notin T_j$).

Proof of correctness of Dijkstra, continued

- WLOG assume R and Q are different and let edge $f = vz$ be the first edge in path R that is not in T_j.
- Assume also that $v \in V(T_j)$ (and hence $z \notin T_j$).
- Let K be the subpath of R from z to y.

Proof of correctness of Dijkstra, continued

- WLOG assume R and Q are different and let edge $f = vz$ be the first edge in path R that is not in T_j.
- Assume also that $v \in V(T_j)$ (and hence $z \notin T_j$).
- Let K be the subpath of R from z to y.
- Since e was a frontier edge used in the $(j+1)$st iteration of Algorithm Dijkstra, we have $P(e) \leq P(f)$. Why?

Dijkstra proof, continued: $P(e) \leq P(f)$

- In that case, $length(R) = dist[v] + \omega(f) + length(K)$

Dijkstra proof, continued: $P(e) \leq P(f)$

- In that case, $length(R) = dist[v] + \omega(f) + length(K)$
- $= P(f) + length(K)$

Dijkstra proof, continued: $P(e) \leq P(f)$

- In that case, $length(R) = dist[v] + \omega(f) + length(K)$
- $= P(f) + length(K)$
- $\geq P(e) \geq length(Q)$.

Dijkstra proof, continued: $P(e) \leq P(f)$

- In that case, $length(R) = dist[v] + \omega(f) + length(K)$
- $= P(f) + length(K)$
- $\geq P(e) \geq length(Q)$.
- In total, we have $length(R) \geq length(Q)$, as desired.

Dijkstra proof, continued: $P(e) \leq P(f)$

- In that case, $length(R) = dist[v] + \omega(f) + length(K)$
- $= P(f) + length(K)$
- $\geq P(e) \geq length(Q)$.

- In total, we have $length(R) \geq length(Q)$, as desired.

- **Conclusion.** For any $j \in \{0, 1, \ldots, |V(G)| - 1\}$ we have that T_j is a shortest path tree for G. **QED**

A Question

Where did we use the fact that the edge weights of G are non-negative?

Final Remarks about Dijkstra

1. **Implementation.** Use a minimum priority queue by way of a binary heap: see section **24.3** for details.

Final Remarks about Dijkstra

1. **Implementation.** Use a minimum priority queue by way of a binary heap: see section **24.3** for details.

2. **BFS tree.** Algorithm Dijkstra gives us a method with which we can construct a **B**readth **F**irst **S**earch (BFS) tree.

BFS Tree

- Assume the edges of graph G all have weight $\mathbf{1}$.

BFS Tree

- Assume the edges of graph G all have weight $\mathbf{1}$.
- Choose a root vertex r and run G through Dijkstra's Algorithm.

BFS Tree

- Assume the edges of graph G all have weight $\mathbf{1}$.
- Choose a root vertex r and run G through Dijkstra's Algorithm.
- **Outcome.** In a Dijkstra tree,

BFS Tree

- Assume the edges of graph G all have weight $\mathbf{1}$.
- Choose a root vertex r and run G through Dijkstra's Algorithm.
- **Outcome.** In a Dijkstra tree,
  1. vertices that are distance 1 from r are at level one,

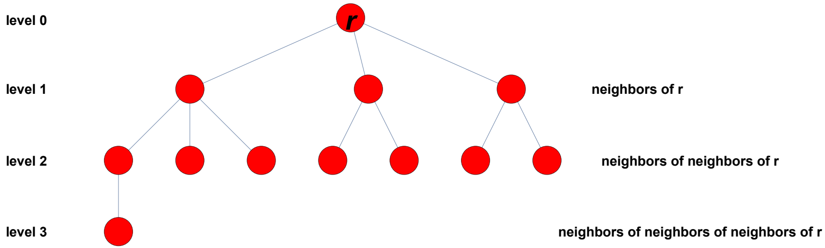
BFS Tree

- Assume the edges of graph G all have weight $\mathbf{1}$.
- Choose a root vertex r and run G through Dijkstra's Algorithm.
- **Outcome.** In a Dijkstra tree,
  1. vertices that are distance 1 from r are at level one,
  2. vertices that are distance 2 from r are at level two,

BFS Tree

- Assume the edges of graph G all have weight $\mathbf{1}$.
- Choose a root vertex r and run G through Dijkstra's Algorithm.
- **Outcome.** In a Dijkstra tree,
  1. vertices that are distance 1 from r are at level one,
  2. vertices that are distance 2 from r are at level two,
  3. ETC

BFS tree, continued



Generalization of Dijkstra: Bellman-Ford Algorithm

- Recall that the input to Dijkstra's algorithms required that the edges are undirected and have non-negative weight.

Generalization of Dijkstra: Bellman-Ford Algorithm

- Recall that the input to Dijkstra's algorithms required that the edges are undirected and have non-negative weight.
- What happens if the weights are allowed to be from all of $\mathbb{R}$ and are directed?

Generalization of Dijkstra: Bellman-Ford Algorithm

- Recall that the input to Dijkstra's algorithms required that the edges are undirected and have non-negative weight.

- What happens if the weights are allowed to be from all of $\mathbb{R}$ and are directed?

- **Set-up.** Let G be a directed graph with edges weighted from $\mathbb{R}$ and let s be the source vertex (i.e., the root).

Generalization of Dijkstra: Bellman-Ford Algorithm

- Recall that the input to Dijkstra's algorithms required that the edges are undirected and have non-negative weight.
- What happens if the weights are allowed to be from all of $\mathbb{R}$ and are directed?
- **Set-up.** Let G be a directed graph with edges weighted from $\mathbb{R}$ and let s be the source vertex (i.e., the root).
- Recall that $\omega(uv)$ is the weight of edge uv and since the edges are directed, order matters.

Generalization of Dijkstra: Bellman-Ford Algorithm

- Recall that the input to Dijkstra's algorithms required that the edges are undirected and have non-negative weight.

- What happens if the weights are allowed to be from all of $\mathbb{R}$ and are directed?

- **Set-up.** Let G be a directed graph with edges weighted from $\mathbb{R}$ and let s be the source vertex (i.e., the root).

- Recall that $\omega(uv)$ is the weight of edge uv and since the edges are directed, order matters.

- One of the main components of the **Bellman-Ford** algorithm is called **Edge Relaxation**.

Edge Relaxation explained

- Let G be any weighted graph (directed or undirected).

Edge Relaxation explained

- Let G be any weighted graph (directed or undirected).
- Define the label $d[u]$ for each $u \in V(G)$, whose purpose is to give

Edge Relaxation explained

- Let G be any weighted graph (directed or undirected).
- Define the label $d[u]$ for each $u \in V(G)$, whose purpose is to give
- a best approximation for the distance from source vertex s to vertex u **so far**.

Edge Relaxation explained

- Let G be any weighted graph (directed or undirected).
- Define the label $d[u]$ for each $u \in V(G)$, whose purpose is to give
- a best approximation for the distance from source vertex s to vertex u **so far**.
- **Initialize** $d[u]$ to ∞ for every vertex $v \neq s$ and set $d[s] = 0$.

Edge Relaxation explained

- Let G be any weighted graph (directed or undirected).
- Define the label $d[u]$ for each $u \in V(G)$, whose purpose is to give
- a best approximation for the distance from source vertex s to vertex u **so far**.
- **Initialize** $d[u]$ to ∞ for every vertex $v \neq s$ and set $d[s] = 0$.
- We gather a collection of vertices C that ultimately become parts of paths from s to u,

Edge Relaxation explained

- Let G be any weighted graph (directed or undirected).
- Define the label $d[u]$ for each $u \in V(G)$, whose purpose is to give
- a best approximation for the distance from source vertex s to vertex u **so far**.
- **Initialize** $d[u]$ to ∞ for every vertex $v \neq s$ and set $d[s] = 0$.
- We gather a collection of vertices C that ultimately become parts of paths from s to u,
- beginning with the neighbors of s, and then the neighbors of the neighbors of s, and so on.

Edge relaxation, continued

- In particular, at each iteration of the algorithm, we choose a vertex $u \notin C$ with the smallest possible $d[u]$ label,

Edge relaxation, continued

- In particular, at each iteration of the algorithm, we choose a vertex $u \notin C$ with the smallest possible $d[u]$ label,
- and then add u to the set C;

Edge relaxation, continued

- In particular, at each iteration of the algorithm, we choose a vertex $u \notin C$ with the smallest possible $d[u]$ label,

- and then add u to the set C;

- then update the label $d[z]$ for all vertices z that are neighbors of u and that are not already in C.

Edge relaxation, continued

- In particular, at each iteration of the algorithm, we choose a vertex $u \notin C$ with the smallest possible $d[u]$ label,
- and then add u to the set C;
- then update the label $d[z]$ for all vertices z that are neighbors of u and that are not already in C.
- The procedure so described allows the algorithm to detect if there is a shorter way to get from s to u by way of z.

Edge relaxation, continued

- In particular, at each iteration of the algorithm, we choose a vertex $u \notin C$ with the smallest possible $d[u]$ label,
- and then add u to the set C;
- then update the label $d[z]$ for all vertices z that are neighbors of u and that are not already in C.
- The procedure so described allows the algorithm to detect if there is a shorter way to get from s to u by way of z.
- This technique is called **edge relaxation**: it takes an old estimate of the distance from s to u, and

Edge relaxation, continued

- In particular, at each iteration of the algorithm, we choose a vertex $u \notin C$ with the smallest possible $d[u]$ label,
- and then add u to the set C;
- then update the label $d[z]$ for all vertices z that are neighbors of u and that are not already in C.
- The procedure so described allows the algorithm to detect if there is a shorter way to get from s to u by way of z.
- This technique is called **edge relaxation**: it takes an old estimate of the distance from s to u, and
- and tries to improve (ie, decrease) the value of the distance label $d[u]$.

Pseudocode for Edge Relaxation

- **if** $d[u] + \omega(uz) < d[z]$ **then**

Pseudocode for Edge Relaxation

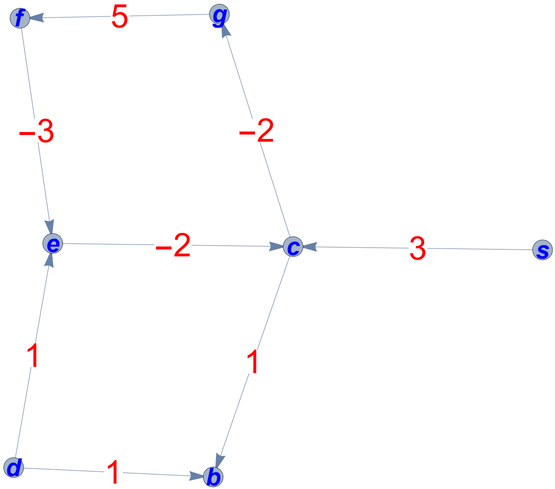
- **if** $d[u] + \omega(uz) < d[z]$ **then**
- $d[z] = d[u] + \omega(uz)$

Reachable negative-weight cycles are bad

- Before giving the Bellman-Ford algorithm, let's look at scenario in a graph for which there is no shortest path tree.

Reachable negative-weight cycles are bad

- Before giving the Bellman-Ford algorithm, let's look at scenario in a graph for which there is no shortest path tree.



-

Negative-weight cycles are bad

- **The Point.** If there exists a negative-weight cycle reachable from source vertex s then there is no shortest path from s to any other vertex reachable from s.

Bellman-Ford Algorithm (**B-F**(G, s))

- **Input.** G, a directed graph and $s \in V(G)$, a source vertex.

Bellman-Ford Algorithm (**B-F**(G, s))

- **Input.** G, a directed graph and $s \in V(G)$, a source vertex.
- **Output.** Either **TRUE** if there is no reachable-from-s negative weight cycle or **FALSE**. The former will also keep track of the shortest path tree.

B-F(G, s), continued

- $d[v] = \infty$ for all $v \in V(G)$
- $d[s] = 0$
- **For** $i = 1$ **to** $|V(G)| - 1$
-     **For** each $uv \in E(G)$
-         **if** $d[u] + \omega(uv) < d[v]$
-         **then** $d[v] = d[u] + \omega(uv)$
-         $parent[v] = u$
-     **For** each edge uv
-         **if** $d[u] + \omega(uv) < d[v]$
-         **then return** "False"
- **return** "True"

The red code is the edge-relaxation part of the algorithm.

Some Analysis of Bellman-Ford

- For a full explanation of the correctness of Bellman-Ford, see chapter 24 in our text.

Some Analysis of Bellman-Ford

- For a full explanation of the correctness of Bellman-Ford, see chapter 24 in our text.
- We'll zoom in on the line

Some Analysis of Bellman-Ford

- For a full explanation of the correctness of Bellman-Ford, see chapter 24 in our text.

- We'll zoom in on the line

- **if** $d[u] + \omega(uv) < d[v]$ **then return** "False"

Some Analysis of Bellman-Ford

- For a full explanation of the correctness of Bellman-Ford, see chapter 24 in our text.
- We'll zoom in on the line
- **if** $d[u] + \omega(uv) < d[v]$ **then return** "False"
- and prove the following claim:

Some Analysis of Bellman-Ford

- For a full explanation of the correctness of Bellman-Ford, see chapter 24 in our text.
- We'll zoom in on the line
- **if** $d[u] + \omega(uv) < d[v]$ **then return** "False"
- and prove the following claim:
- **Claim.** If the input graph G contains a negative weight cycle reachable form s, then $\mathbf{B\text{-}F}(G, s)$ returns "False."

Proof of Claim

- Let $v_0 v_1 \ldots v_k$ with $v_0 = v_k$ be a negative weight cycle.

Proof of Claim

- Let $v_0 v_1 \ldots v_k$ with $v_0 = v_k$ be a negative weight cycle.
- That is $\sum_{i=0}^{k-1} \omega(v_i v_{i+1}) < 0$.

Proof of Claim

- Let $v_0 v_1 \ldots v_k$ with $v_0 = v_k$ be a negative weight cycle.
- That is $\sum_{i=0}^{k-1} \omega(v_i v_{i+1}) < 0$.
- BWOC assume that algorithm **B-F** does not output "False."

Proof of Claim

- Let $v_0 v_1 \ldots v_k$ with $v_0 = v_k$ be a negative weight cycle.
- That is $\sum_{i=0}^{k-1} \omega(v_i v_{i+1}) < 0$.
- BWOC assume that algorithm **B-F** does not output "False."
- Then $d[v_i] + \omega(v_i v_{i+1}) \geq d[v_{i+1}] \ \ \forall i \in \{0, 1, \ldots, k-1\}$.

Proof of Claim

- Let $v_0 v_1 \dots v_k$ with $v_0 = v_k$ be a negative weight cycle.
- That is $\sum_{i=0}^{k-1} \omega(v_i v_{i+1}) < 0$.
- BWOC assume that algorithm **B-F** does not output "False."
- Then $d[v_i] + \omega(v_i v_{i+1}) \geq d[v_{i+1}] \;\; \forall i \in \{0, 1, \dots, k-1\}$.
- $\Rightarrow \sum_{i=0}^{k-1} d[v_i] + \sum_{i=0}^{k-1} \omega(v_i v_{i+1}) \geq \sum_{i=0}^{k-1} d[v_{i+1}]$.

Proof of Claim

- Let $v_0 v_1 \ldots v_k$ with $v_0 = v_k$ be a negative weight cycle.
- That is $\sum_{i=0}^{k-1} \omega(v_i v_{i+1}) < 0$.
- BWOC assume that algorithm **B-F** does not output "False."
- Then $d[v_i] + \omega(v_i v_{i+1}) \geq d[v_{i+1}] \;\; \forall i \in \{0, 1, \ldots, k-1\}$.
- $\Rightarrow \sum_{i=0}^{k-1} d[v_i] + \sum_{i=0}^{k-1} \omega(v_i v_{i+1}) \geq \sum_{i=0}^{k-1} d[v_{i+1}]$.
- Observe that $\sum_{i=0}^{k-1} d[v_i] = \sum_{i=0}^{k-1} d[v_{i+1}]$. Why?

Proof of Claim

- Let $v_0 v_1 \ldots v_k$ with $v_0 = v_k$ be a negative weight cycle.
- That is $\sum_{i=0}^{k-1} \omega(v_i v_{i+1}) < 0$.
- BWOC assume that algorithm **B-F** does not output "False."
- Then $d[v_i] + \omega(v_i v_{i+1}) \geq d[v_{i+1}] \;\; \forall i \in \{0, 1, \ldots, k-1\}$.
- $\Rightarrow \sum_{i=0}^{k-1} d[v_i] + \sum_{i=0}^{k-1} \omega(v_i v_{i+1}) \geq \sum_{i=0}^{k-1} d[v_{i+1}]$.
- Observe that $\sum_{i=0}^{k-1} d[v_i] = \sum_{i=0}^{k-1} d[v_{i+1}]$. Why?
- Thus $\sum_{i=0}^{k-1} \omega(v_i v_{i+1}) \geq 0$, which is the desired contradiction.

Proof of Claim

- Let $v_0 v_1 \ldots v_k$ with $v_0 = v_k$ be a negative weight cycle.
- That is $\sum_{i=0}^{k-1} \omega(v_i v_{i+1}) < 0$.
- BWOC assume that algorithm **B-F** does not output "False."
- Then $d[v_i] + \omega(v_i v_{i+1}) \geq d[v_{i+1}] \;\; \forall i \in \{0, 1, \ldots, k-1\}$.
- $\Rightarrow \sum_{i=0}^{k-1} d[v_i] + \sum_{i=0}^{k-1} \omega(v_i v_{i+1}) \geq \sum_{i=0}^{k-1} d[v_{i+1}]$.
- Observe that $\sum_{i=0}^{k-1} d[v_i] = \sum_{i=0}^{k-1} d[v_{i+1}]$. Why?
- Thus $\sum_{i=0}^{k-1} \omega(v_i v_{i+1}) \geq 0$, which is the desired contradiction.
- **QED**

Runtime of **B-F**

- Assume the input graph G is simple.

Runtime of **B-F**

- Assume the input graph G is simple.
- Then the runtime of algorithm **B-F** is $O(|V(G)|^3)$.

Runtime of **B-F**

- Assume the input graph G is simple.
- Then the runtime of algorithm **B-F** is $O(|V(G)|^3)$.
- **Proof:** Exercise.

Runtime of **B-F**

- Assume the input graph G is simple.
- Then the runtime of algorithm **B-F** is $O(|V(G)|^3)$.
- **Proof:** Exercise.
- Finally, a good alternative source for both Dijkstra and Bellman-Ford is Gross and Yellen's textbook **Graph Theory and Applications**, especially for the proof of Dijkstra and the use of frontier edges in tree growing.

New Topic in Graph Algorithms: Network Flows

- We'll begin with some definitions.

New Topic in Graph Algorithms: Network Flows

- We'll begin with some definitions.
- A **flow network** is a directed graph G in which every edge $uv \in E(G)$ has a positive **capacity** $c(uv)$, and

New Topic in Graph Algorithms: Network Flows

- We'll begin with some definitions.
- A **flow network** is a directed graph G in which every edge $uv \in E(G)$ has a positive **capacity** $c(uv)$, and
- furthermore $c(uv) = 0$ if uv is not an edge.

New Topic in Graph Algorithms: Network Flows

- We'll begin with some definitions.
- A **flow network** is a directed graph G in which every edge $uv \in E(G)$ has a positive **capacity** $c(uv)$, and
- furthermore $c(uv) = 0$ if uv is not an edge.
- In other words, the function c describes the weights on the edges and non-edges of G.

New Topic in Graph Algorithms: Network Flows

- We'll begin with some definitions.
- A **flow network** is a directed graph G in which every edge $uv \in E(G)$ has a positive **capacity** $c(uv)$, and
- furthermore $c(uv) = 0$ if uv is not an edge.
- In other words, the function c describes the weights on the edges and non-edges of G.
- There is a **source** vertex s and a **sink** vertex t.

New Topic in Graph Algorithms: Network Flows

- We'll begin with some definitions.
- A **flow network** is a directed graph G in which every edge $uv \in E(G)$ has a positive **capacity** $c(uv)$, and
- furthermore $c(uv) = 0$ if uv is not an edge.
- In other words, the function c describes the weights on the edges and non-edges of G.
- There is a **source** vertex s and a **sink** vertex t.
- A **flow** is a real-valued function $f : V \times V \to \mathbb{R}$ that has the following properties:

Network flow definitions and constraints, continued

- A **flow** is a real-valued function $f : V \times V \to \mathbb{R}$ that has the following three properties:

Network flow definitions and constraints, continued

- A **flow** is a real-valued function $f : V \times V \to \mathbb{R}$ that has the following three properties:

  1. **Capacity Constraint:** $f(uv) \leq c(uv) \quad \forall u, v \in V(G)$. "can't burst the pipes"

Network flow definitions and constraints, continued

- A **flow** is a real-valued function $f : V \times V \to \mathbb{R}$ that has the following three properties:

  1. **Capacity Constraint:** $f(uv) \leq c(uv) \quad \forall u, v \in V(G)$. "can't burst the pipes"
  2. **Skew Symmetry:** $f(uv) = -f(vu)$. "positive flow in one direction = negative flow in the opposite direction."

Network flow definitions and constraints, continued

- A **flow** is a real-valued function $f : V \times V \to \mathbb{R}$ that has the following three properties:

  1. **Capacity Constraint:** $f(uv) \leq c(uv) \quad \forall u, v \in V(G)$. "can't burst the pipes"
  2. **Skew Symmetry:** $f(uv) = -f(vu)$. "positive flow in one direction = negative flow in the opposite direction."
  3. **Flow Conservation:** $\sum_{v \in V(G)} f(uv) = 0 \quad \forall v \in V(G) \setminus \{s, t\}$. "amount of water entering u = amount of water leaving u"

Network flow definitions and constraints, continued

- A **flow** is a real-valued function $f : V \times V \to \mathbb{R}$ that has the following three properties:

  1. **Capacity Constraint:** $f(uv) \leq c(uv) \quad \forall u, v \in V(G)$. "can't burst the pipes"
  2. **Skew Symmetry:** $f(uv) = -f(vu)$. "positive flow in one direction = negative flow in the opposite direction."
  3. **Flow Conservation:** $\sum_{v \in V(G)} f(uv) = 0 \quad \forall v \in V(G) \setminus \{s, t\}$. "amount of water entering u = amount of water leaving u"

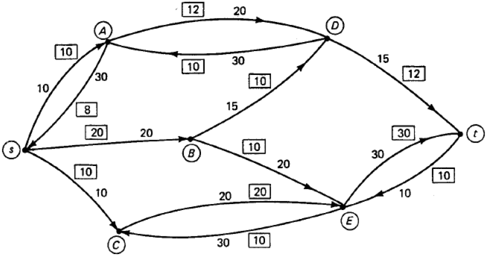
- Finally, the **value of flow** f is $\sum_{v \in V(G) \setminus s} f(sv)$. That is, the flow is the amount of water leaving the source vertex s.

Example from Herb Wilf's downloadable Algorithms text

- The network flow example below is from Chapter 3 of `https://www.math.upenn.edu/~wilf/AlgComp3.html`

Example from Herb Wilf's downloadable Algorithms text

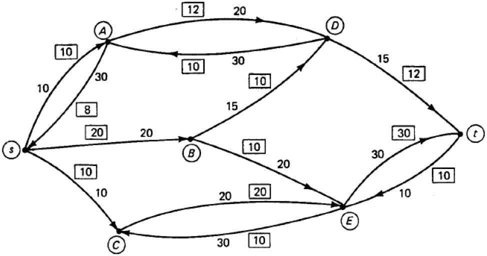
- The network flow example below is from Chapter 3 of `https://www.math.upenn.edu/~wilf/AlgComp3.html`

- 

- The number in the □ is the flow and can change.

- The number in the non-box is the capacity and can't change.

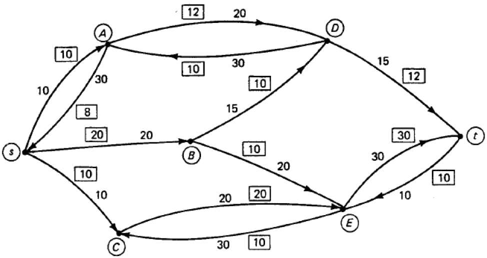
Example from Herb Wilf's downloadable Algorithms text

- The network flow example below is from Chapter 3 of `https://www.math.upenn.edu/~wilf/AlgComp3.html`
- 

- The number in the □ is the flow and can change.
- The number in the non-box is the capacity and can't change.
- **Exercise.** Check that this network satisfies skew symmetry, the capacity constraint, and flow conservation.

Example from Herb Wilf's downloadable Algorithms text

- The network flow example below is from Chapter 3 of `https://www.math.upenn.edu/~wilf/AlgComp3.html`

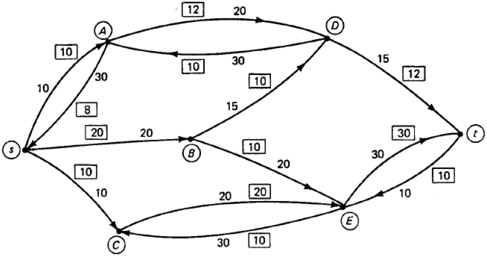
- 

- The number in the $\square$ is the flow and can change.

- The number in the non-box is the capacity and can't change.

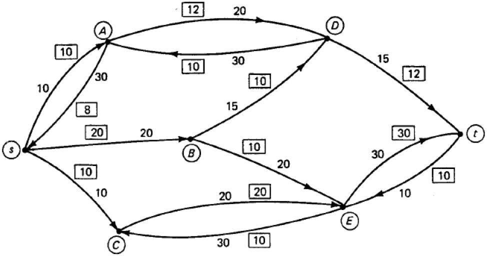
- **Exercise.** Check that this network satisfies skew symmetry, the capacity constraint, and flow conservation.

Example from Herb Wilf's downloadable Algorithms text



-

Example from Herb Wilf's downloadable Algorithms text

- 

- **Exercise.** What is the flow of the above network?

Hmmmm

- We haven't stated a problem yet!!

Hmmmm

- We haven't stated a problem yet!!
- **Here goes.**

Hmmmm

- We haven't stated a problem yet!!
- **Here goes.**
- **The Network Flow Problem.** Given a flow network G with capacity constraint function c, find the flow of maximum value.

Source for applications

- For one, of many, interesting sources for applications of the network flow problem, see

Source for applications

- For one, of many, interesting sources for applications of the network flow problem, see
- `http://blogs.cornell.edu/info2040/2012/11/04/applications-of-network-flow/`

Source for applications

- For one, of many, interesting sources for applications of the network flow problem, see
- `http://blogs.cornell.edu/info2040/2012/11/04/applications-of-network-flow/`
- **Next Week.** More on network flows by way of a side-trip to linear programming, the max-flow/min cut theorem, and the Ford-Fulkerson hall of fame.