Advanced Algorithmics (6EAP)  
Graphs II

Jaak Vilo  
2011 Spring
WEIGHTED GRAPH ALGORITHMS
Weighted Graph Algorithms

Beyond DFS/BFS exists an alternate universe of algorithms for *edge-weighted graphs*. Our adjacency list representation quietly supported these graphs:

```c
typedef struct {
    int y;
    int weight;
    struct edgenode *next;
} edgenode;
```
Minimum Spanning Tree

• Definition: Given an undirected graph, and for each edge \((v, u) \in E\), we have a weight \(w(u, v)\) specifying the cost to connect \(u\) and \(v\). Find an acyclic subset \(T \subseteq E\) that connects all of the vertices and whose total weight is minimized
  \[ w(T) = \sum_{(u,v)\in T} w(u, v) \]
  – May have more than one MST with the same weight

• Two classic algorithms: \(O(E \lg V) \Rightarrow\) Greedy Algorithms
  – Kruskal’s algorithm
  – Prim’s algorithm
Minimum Spanning Trees

A tree is a connected graph with no cycles. A spanning tree is a subgraph of $G$ which has the same set of vertices of $G$ and is a tree.

A minimum spanning tree of a weighted graph $G$ is the spanning tree of $G$ whose edges sum to minimum weight. There can be more than one minimum spanning tree in a graph → consider a graph with identical weight edges.
Equal weights in left graph (a)
Why Minimum Spanning Trees?

The minimum spanning tree problem has a long history – the first algorithm dates back at least to 1926!. Minimum spanning tree is always taught in algorithm courses since (1) it arises in many applications, (2) it is an important example where greedy algorithms always give the optimal answer, and (3) Clever data structures are necessary to make it work.

In greedy algorithms, we make the decision of what next to do by selecting the best local option from all available choices – without regard to the global structure.
Applications of Minimum Spanning Trees

Minimum spanning trees are useful in constructing networks, by describing the way to connect a set of sites using the smallest total amount of wire. Minimum spanning trees provide a reasonable way for clustering points in space into natural groups. What are natural clusters in the friendship graph?
Minimum Spanning Trees and TSP

When the cities are points in the Euclidean plane, the minimum spanning tree provides a good heuristic for traveling salesman problems. The optimum traveling salesman tour is at most twice the length of the minimum spanning tree.

The Optimal Traveling System tour is at most twice the length of the minimum spanning tree.

Note: There can be more than one minimum spanning tree considered as a group with identical weight edges.
Growing a Minimum Spanning Tree (MST)

• Generic algorithm
  – Grow MST one edge at a time
  – Manage a set of edges A, maintaining the following loop invariant:
    • Prior to each iteration, A is a subset of some MST
  – At each iteration, we determine an edge (u, v) that can be added to A without violating this invariant
    • A \cup \{(u, v)\} is also a subset of a MST
    • (u, v) is called a \textit{safe edge} for A
GENERIC-MST

**GENERIC-MST** *(G, w)*

1. \( A \leftarrow \emptyset \)
2. while \( A \) does not form a spanning tree
3. \hspace{1cm} do find an edge \((u, v)\) that is safe for \( A \)
4. \hspace{1cm} \( A \leftarrow A \cup \{(u, v)\} \)
5. return \( A \)

- Loop in lines 2-4 is executed \(|V| - 1\) times
  - Any MST tree contains \(|V| - 1\) edges
  - The execution time depends on how to find a safe edge
How to Find A Safe Edge?

• **Theorem.** Let $A$ be a subset of $E$ that is included in some MST, let $(S, V-S)$ be any cut of $G$ that respects $A$, and let $(u, v)$ be a light edge crossing $(S, V-S)$. Then edge $(u, v)$ is safe for $A$
  
  – **Cut** $(S, V-S)$: a partition of $V$
  – **Crossing edge**: one endpoint in $S$ and the other in $V-S$
  – A cut respects a set of $A$ of edges if no edges in $A$ crosses the cut
  – A light edge crossing a cut if its weight is the minimum of any edge crossing the cut
Figure 23.2  Two ways of viewing a cut \((S, V - S)\) of the graph from Figure 23.1. (a) The vertices in the set \(S\) are shown in black, and those in \(V - S\) are shown in white. The edges crossing the cut are those connecting white vertices with black vertices. The edge \((d, c)\) is the unique light edge crossing the cut. A subset \(A\) of the edges is shaded; note that the cut \((S, V - S)\) respects \(A\), since no edge of \(A\) crosses the cut. (b) The same graph with the vertices in the set \(S\) on the left and the vertices in the set \(V - S\) on the right. An edge crosses the cut if it connects a vertex on the left with a vertex on the right.
Illustration of Theorem 23.1

- $A=\{(a,b), (c, i), (h, g), \{g, h\}\}$
- $S=\{a, b, c, i, e\}; \ V-S = \{h, g, f, d\} \quad \Rightarrow \text{many kinds of cuts satisfying the requirements of Theorem 23.1}$

- $(c, f)$ is the light edges crossing $S$ and $V-S$ and will be a safe edge
Proof of Theorem 23.1

• Let $T$ be a MST that includes $A$, and assume $T$ does not contain the light edge $(u, v)$, since if it does, we are done.

• Construct another MST $T'$ that includes $A \cup \{(u, v)\}$ from $T$
  
  – Next slide
  
  – $T' = T - \{(x, y) \cup (u, v)\}$
  
  – $T'$ is also a MST since $W(T') = W(T) - w(x, y) + w(u, v) \leq W(T)$

• $(u, v)$ is actually a safe edge for $A$
  
  – Since $A \subseteq T$ and $(x, y) \notin A \Rightarrow A \subseteq T'$
  
  – $\Rightarrow A \cup \{(u, v)\} \subseteq T'$
Figure 23.3  The proof of Theorem 23.1. The vertices in $S$ are black, and the vertices in $V - S$ are white. The edges in the minimum spanning tree $T$ are shown, but the edges in the graph $G$ are not. The edges in $A$ are shaded, and $(u, v)$ is a light edge crossing the cut $(S, V - S)$. The edge $(x, y)$ is an edge on the unique path $p$ from $u$ to $v$ in $T$. A minimum spanning tree $T'$ that contains $(u, v)$ is formed by removing the edge $(x, y)$ from $T$ and adding the edge $(u, v)$. 
Properties of GENERIC-MST

• As the algorithm proceeds, the set $A$ is always acyclic
• $G_A=(V, A)$ is a forest, and each of the connected component of $G_A$ is a tree
• Any safe edge $(u, v)$ for $A$ connects distinct component of $G_A$, since $A \cup \{(u, v)\}$ must be acyclic
• Corollary 23.2. Let $A$ be a subset of $E$ that is included in some MST, and let $C = (V_C, E_C)$ be a connected components (tree) in the forest $G_A=(V, A)$. If $(u, v)$ is a light edge connecting $C$ to some other components in $G_A$, then $(u, v)$ is safe for $A$
The Algorithms of Kruskal and Prim

- Kruskal’s Algorithm
  - A is a forest
  - The safe edge added to A is always a least-weight edge in the graph that connects two distinct components

- Prim’s Algorithm
  - A forms a single tree
  - The safe edge added to A is always a least-weight edge connecting the tree to a vertex not in the tree
Prim’s Algorithm

• The edges in the set A always forms a single tree
• The tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V
• At each step, a light edge is added to the tree A that connects A to an isolated vertex of $G_A = (V, A)$
• Greedy since the tree is augmented at each step with an edge that contributes the minimum amount possible to the tree’s weight
Prim’s Algorithm

If $G$ is connected, every vertex will appear in the minimum spanning tree. If not, we can talk about a minimum spanning forest.

Prim’s algorithm starts from one vertex and grows the rest of the tree an edge at a time.

As a greedy algorithm, which edge should we pick? The cheapest edge with which can grow the tree by one vertex without creating a cycle.
Prim’s Algorithm (Pseudocode)

During execution each vertex $v$ is either in the tree, fringe (meaning there exists an edge from a tree vertex to $v$) or unseen (meaning $v$ is more than one edge away).

Prim-MST(G)

- Select an arbitrary vertex $s$ to start the tree from.
- While (there are still non-tree vertices)
  - Select the edge of minimum weight between a tree and node
  - Add the selected edge and vertex to the tree $T_{\text{prim}}$.

This creates a spanning tree, since no cycle can be introduced, but is it minimum?
Prim’s Algorithm in Action

G

Prim(G, A)

Kruskal(G)
Key idea of Prim’s algorithm

Select a vertex to be a tree-node

while (there are non-tree vertices)
{
    if (there is no edge connecting a tree node with a non-tree node)
        return “no spanning tree”

    select an edge of minimum weight between a tree node and a non-tree node

    add the selected edge and its new vertex to the tree

} return tree
Prim’s Algorithm (Cont.)

• How to efficiently select the safe edge to be added to the tree?
  – Use a min-priority queue $Q$ that stores all vertices not in the tree
    • Based on $\text{key}[v]$, the minimum weight of any edge connecting $v$ to a vertex in the tree
      – $\text{Key}[v] = \infty$ if no such edge
  
• $\pi[v] = \text{parent of } v \text{ in the tree}$

• $A = \{(v, \pi[v]): v \in V - \{r\} - Q\} \Rightarrow \text{finally } Q = \text{empty}$
Prim’s Algorithm

1. for each $u \in V$
2. do $D[u] \leftarrow \infty$
3. $D[r] \leftarrow 0$
4. $MH \leftarrow \text{make-heap}(D,V, \emptyset)$//No edges
5. $T \leftarrow \emptyset$
6.
7. while $MH \neq \emptyset$ do
8. $(u,e) \leftarrow MH\.extractMin()$
9. add $(u,e)$ to $T$
10. for each $v \in \text{Adjacent}(u)$
11. do if $v \in MH \&\& w(u,v) < D[v]$
12. then $D[v] \leftarrow w(u,v)$
13. MH\.decreaseDistance($D[v], v, (u,v)$)
14. return $T$ // $T$ is a MST

Lines 1-5 initialize the min-heap (MH) to contain all vertices.
Distances for all vertices, except $r$, are set to infinity.
$r$ is the starting vertex of the $T$
The $T$ so far is empty

Add the closest vertex and edge to current $T$
Get all adjacent vertices $v$ of $u$, update $D$ of each non-tree vertex adjacent to $u$
Store the current minimum weight edge and updated distance in the MH
MST-PRIM\( (G, w, r) \)

1. for each \( u \in V[G] \)
2. \hspace{1em} do \( \text{key}[u] \leftarrow \infty \)
3. \hspace{1em} \( \pi[u] \leftarrow \text{NIL} \)
4. \( \text{key}[r] \leftarrow 0 \)
5. \( Q \leftarrow V[G] \)
6. while \( Q \neq \emptyset \)
7. \hspace{1em} do \( u \leftarrow \text{EXTRACT-MIN}(Q) \)
8. \hspace{1em} for each \( v \in \text{Adj}[u] \)
9. \hspace{2em} do if \( v \in Q \) and \( w(u, v) < \text{key}[v] \)
10. \hspace{3em} then \( \pi[v] \leftarrow u \)
11. \hspace{3em} \( \text{key}[v] \leftarrow w(u, v) \)
Illustration of MST-PRIM

\( u=a, \text{adj}[a] = \{b, h\} \)
\( \pi[b] = \pi[h] = a \)
\( \text{key}[b] = 4; \text{key}[h] = 8 \)

\( u=b, \text{adj}[b] = \{a, c, h\} \)
\( \pi[h] = a ; \pi[c] = b \)
\( \text{key}[h] = 8; \text{key}[c] = 8 \)

\( u=c, \text{adj}[c] = \{b, i, d, f\} \)
\( \pi[d] = \pi[i] = \pi[f] = c \)
\( \text{key}[h] = 8; \text{key}[d] = 7; \text{key}[i] = 2; \text{key}[f] = 4 \)
Properties of MST-PRIM

• Prior to each iteration of the while loop of lines 6—11
  – A = {(v, π[v]): v ∈ V-{r}-Q}
  – The vertices already placed into the MST are those in V-Q
  – For all vertices v ∈ Q, if π[v] ≠ NIL, then key[v] < ∞ and key [v] is the weight of a light edge (v, π[v]) connecting v to some vertex already placed into the MST

• Line 7: identify a vertex u ∈ Q incident on a light edge crossing (V-Q, Q) → add u to V-Q and (u, π[u]) to A

• Lines 8—11: update key and π of every vertex v adjacent to u but not in the tree
Performance of MST-PRIM

• Use binary min-heap to implement the min-priority queue Q
  – BUILD-MIN-HEAP (line 5): \(O(V)\)
  – The body of while loop is executed \(|V|\) times
    • EXTRACT-MIN: \(O(\lg V)\)
  – The for loop in lines 8-11 is executed \(O(E)\) times altogether
    • Line 11: DECREASE-KEY operation: \(O(\lg V)\)
  – Total performance = \(O(V \lg V + E \lg V) = O(E \lg V)\)

• Use Fibonacci heap to implement the min-priority queue Q
  – \(O(E + V \lg V)\)
Why is Prim Correct?

We use a proof by contradiction:
Suppose Prim’s algorithm does not always give the minimum cost spanning tree on some graph.
If so, there is a graph on which it fails.
And if so, there must be a first edge $\langle x, y \rangle$ Prim adds such that the partial tree $V'$ cannot be extended into a minimum spanning tree.
But if \((x, y)\) is not in \(MST(G)\), then there must be a path in \(MST(G)\) from \(x\) to \(y\) since the tree is connected. Let \((v, w)\) be the first edge on this path with one edge in \(V'\). Replacing it with \((x, y)\) we get a spanning tree. with smaller weight, since \(W(v, w) > W(x, y)\). Thus you did not have the MST!!
Kruskal’s Algorithm

Since an easy lower bound argument shows that every edge must be looked at to find the minimum spanning tree, and the number of edges $m = O(n^2)$, Prim’s algorithm is optimal in the worst case. Is that all she wrote? The complexity of Prim’s algorithm is independent of the number of edges. Can we do better with sparse graphs? Yes! Kruskal’s algorithm is also greedy. It repeatedly adds the smallest edge to the spanning tree that does not create a cycle.
Why is Kruskal’s algorithm correct?

Again, we use proof by contradiction. Suppose Kruskal’s algorithm does not always give the minimum cost spanning tree on some graph. If so, there is a graph on which it fails. And if so, there must be a first edge \((x, y)\) Kruskal adds such that the set of edges cannot be extended into a minimum spanning tree.

When we added \((x, y)\) there previously was no path between \(x\) and \(y\), or it would have created a cycle. Thus if we add \((x, y)\) to the optimal tree it must create a cycle. At least one edge in this cycle must have been added after \((x, y)\), so it must have a heavier weight.

Deleting this heavy edge leave a better MST than the optimal tree? A contradiction!
How fast is Kruskal’s algorithm?

What is the simplest implementation?

- Sort the \( m \) edges in \( O(m \lg m) \) time.

- For each edge in order, test whether it creates a cycle the forest we have thus far built — if so discard, else add to forest. With a BFS/DFS, this can be done in \( O(n) \) time (since the tree has at most \( n \) edges).

The total time is \( O(mn) \), but can we do better?
Fast Component Tests Give Fast MST

Kruskal’s algorithm builds up connected components. Any edge where both vertices are in the same connected component create a cycle. Thus if we can maintain which vertices are in which component fast, we do not have test for cycles!

- *Same component*(\(v_1, v_2\)) − Do vertices \(v_1\) and \(v_2\) lie in the same connected component of the current graph?

- *Merge components*(\(C_1, C_2\)) − Merge the given pair of connected components into one component.
Fast Kruskal Implementation

Put the edges in a heap
\[ \text{count} = 0 \]
while \((\text{count} < n - 1)\) do
  get next edge \((v, w)\)
  if \((\text{component}(v) \neq \text{component}(w))\)
    add to \(T\)
    \(\text{component}(v) = \text{component}(w)\)

If we can test components in \(O(\log n)\), we can find the MST in \(O(m \log m)\)!

**Question:** Is \(O(m \log n)\) better than \(O(m \log m)\)?
Union-Find Programs

We need a data structure for maintaining sets which can test if two elements are in the same and merge two sets together. These can be implemented by union and find operations, where

- $\text{Find}(i)$ — Return the label of the root of tree containing element $i$, by walking up the parent pointers until there is no where to go.

- $\text{Union}(i,j)$ — Link the root of one of the trees (say containing $i$) to the root of the tree containing the other (say $j$) so $\text{find}(i)$ now equals $\text{find}(j)$.

See the lecture on trees...
This path compression will let us do better than $O(n \log n)$ for $n$ union-finds.

$O(n)$? Not quite ... Difficult analysis shows that it takes $O(n\alpha(n))$ time, where $\alpha(n)$ is the inverse Ackerman function and $\alpha(\text{number of atoms in the universe}) = 5$. 
Problem of the Day

Suppose we are given the minimum spanning tree $T$ of a given graph $G$ (with $n$ vertices and $m$ edges) and a new edge $e = (u, v)$ of weight $w$ that we will add to $G$. Give an efficient algorithm to find the minimum spanning tree of the graph $G + e$. Your algorithm should run in $O(n)$ time to receive full credit, although slower but correct algorithms will receive partial credit.
Prim vs Kruskal vs Boruvka

**Figure 20.10**
PFS implementation of Prim’s MST algorithm

With PFS, Prim’s algorithm processes just the vertices and edges closest to the MST (in gray).

**Figure 20.13**
Kruskal’s MST algorithm

This sequence shows 1/4, 1/2, 3/4, and the full MST as it evolves.

**Figure 20.16**
Boruvka’s MST algorithm

The MST evolves in just four stages for this example (top to bottom).
Table 20.1 Cost of MST algorithms

This table summarizes the cost (worst-case running time) of various MST algorithms considered in this chapter. The formulas are based on the assumptions that an MST exists (which implies that $E$ is no smaller than $V - 1$) and that there are $X$ edges not longer than the longest edge in the MST (see Property 20.10). These worst-case bounds may be too conservative to be useful in predicting performance on real graphs. The algorithms run in near-linear time in a broad variety of practical situations.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>worst-case cost</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prim (standard)</td>
<td>$V^2$</td>
<td>optimal for dense graphs</td>
</tr>
<tr>
<td>Prim (PFS, heap)</td>
<td>$E \lg V$</td>
<td>conservative upper bound</td>
</tr>
<tr>
<td>Prim (PFS, $d$-heap)</td>
<td>$E \log_d V$</td>
<td>linear unless extremely sparse</td>
</tr>
<tr>
<td>Kruskal</td>
<td>$E \lg E$</td>
<td>sort cost dominates</td>
</tr>
<tr>
<td>Kruskal (partial sort)</td>
<td>$E + X \lg V$</td>
<td>cost depends on longest edge</td>
</tr>
<tr>
<td>Boruvka</td>
<td>$E \lg V$</td>
<td>conservative upper bound</td>
</tr>
</tbody>
</table>
SINGLE-SOURCE SHORTEST PATHS (CHAPTER 24)
Shortest Paths

Finding the shortest path between two nodes in a graph arises in many different applications:

- Transportation problems – finding the cheapest way to travel between two locations.
- Motion planning – what is the most natural way for a cartoon character to move about a simulated environment.
- Communications problems – how long will it take for a message to get between two places? Which two locations are furthest apart, i.e. what is the diameter of the network.
Example: Predictive Mobile text Entry Messaging...

What was the message?
INPUT

Blank Recognition

Candidate Construction

Sentence Disambiguating

OUTPUT

GIVE ME A RING.
Weighting the Graph

The weight of each edge is a function of the probability that these two words will be next to each other in a sentence. ‘hive me’ would be less than ‘give me’, for example. The final system worked extremely well – identifying over 99% of characters correctly based on grammatical and statistical constraints.
Problem Definition

• Given a weighted, directed graph $G=(V, E)$ with weight function $w: E \rightarrow \mathbb{R}$. The weight of path $p=<v_0, v_1, ..., v_k>$ is the sum of the weights of its constituent edges:
  
  
  $$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

• We define the shortest-path weight from $u$ to $v$ by
  
  $$\delta(u, v) = \begin{cases} 
\min \{w(p): u \xrightarrow{p} v\} & \text{If there is a path from } u \text{ to } v, \\
\infty & \text{Otherwise.}
\end{cases}$$

• A shortest path from vertex $u$ to vertex $v$ is then defined as any path with $w(p) = \delta(u, v)$
Variants

• Single-source shortest paths problem – greedy
  – Finds all the shortest path of vertices reachable from a single source vertex s

• Single-destination shortest-path problem
  – By reversing the direction of each edge in the graph, we can reduce this problem to a single-source problem

• Single-pair shortest-path problem
  – No algorithm for this problem are known that run asymptotically faster than the best single-source algorithm in the worst case

• All-pairs shortest-path problem – dynamic programming
  – Can be solved faster than running the single-source shortest-path problem for each vertex
Figure 24.2 (a) A weighted, directed graph with shortest-path weights from source \(s\). (b) The shaded edges form a shortest-paths tree rooted at the source \(s\). (c) Another shortest-paths tree with the same root.
Optimal Substructure of A Shortest-Path

• Lemma 24.1 (Subpath of shortest paths are shortest paths). Let $p=<v_1, v_2, ..., v_k>$ be a shortest path from vertex $v_1$ to $v_k$, and for any $i$ and $j$ such that $1 \leq i \leq j \leq k$, let $p_{ij} = <v_{1i}, v_2, ..., v_j>$ be the subpath of $p$ from vertex $v_i$ to $v_j$. Then $p_{ij}$ is a shortest path from vertex $v_i$ to $v_j$. 
Negative-Weight Edges and Cycles

• Cannot contain a negative-weight cycle
• Of course, a shortest path cannot contain a positive-weight cycle
Relaxation

• For each vertex $v \in V$, we maintain an attribute $d[v]$, which is an upper bound on the weight of a shortest path from source $s$ to $v$. We call $d[v]$ a shortest-path estimate.

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex $v \in V[G]$
2      do $d[v] \leftarrow \infty$
3      $\pi[v] \leftarrow NIL$
4  $d[s] \leftarrow 0$
```
Relaxation (Cont.)

- Relaxing an edge \((u, v)\) consists of testing whether we can improve the shortest path found so far by going through \(u\) and, if so, update \(d[v]\) and \(\pi[v]\)

\[
\begin{align*}
\text{RELAX}(u, v, w) & \\
1 & \text{if } d[v] > d[u] + w(u, v) \\
2 & \text{then } d[v] \leftarrow d[u] + w(u, v) \\
3 & \pi[v] \leftarrow u
\end{align*}
\]

**Figure 24.3** Relaxation of an edge \((u, v)\) with weight \(w(u, v) = 2\). The shortest-path estimate of each vertex is shown within the vertex. (a) Because \(d[v] > d[u] + w(u, v)\) prior to relaxation, the value of \(d[v]\) decreases. (b) Here, \(d[v] \leq d[u] + w(u, v)\) before the relaxation step, and so \(d[v]\) is unchanged by relaxation.
Bellman-Ford

Bellman-Ford ( G, w, s )

1  Initialise-Single-Source(G,S)

2  for i=1 to |G.V|-1

3     for each edge (u,v) ∈ G.E

4         RELAX( u, v, w )

5     for each edge (u,v) ∈ G.E

6         if v.d > u.d + w(u,v)

7         return FALSE

8     return TRUE
Figure 24.4  The execution of the Bellman-Ford algorithm. The source is vertex $s$. The $d$ values are shown within the vertices, and shaded edges indicate predecessor values: if edge $(u, v)$ is shaded, then $\pi[v] = u$. In this particular example, each pass relaxes the edges in the order $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$. (a) The situation just before the first pass over the edges. (b)–(e) The situation after each successive pass over the edges. The $d$ and $\pi$ values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.
Bellman-Ford

- $O(VE)$
Shortest paths on a DAG

1. DAG-Shortest-path(G,w,s)
2. topologically sort vertices
3. Initialise-single-source(G,s)
4. for each vertex u in topological order
5. for each vertex v ∈ G.Adj[u]
6. RELAX(u,v,w)

O(V + E)
Source s
Dijkstra’s Algorithm

• Solve the single-source shortest-paths problem on a weighted, directed graph and all edge weights are nonnegative

• Data structure
  – S: a set of vertices whose final shortest-path weights have already been determined
  – Q: a min-priority queue keyed by their d values

• Idea
  – Repeatedly select the vertex \( u \in V - S \) (kept in Q) with the minimum shortest-path estimate, add \( s u \) to \( S \), and relaxes all edges leaving \( u \)
Dijkstra’s Algorithm (Cont.)

\[
\text{DIJKSTRA}(G, w, s)
\]

1. \text{INITIALIZE-SINGLE-SOURCE}(G, s)
2. \( S \leftarrow \emptyset \)
3. \( Q \leftarrow V[G] \)
4. \textbf{while} \( Q \neq \emptyset \)
5. \hspace{1em} \textbf{do} \( u \leftarrow \text{EXTRACT-MIN}(Q) \)
6. \hspace{2em} \( S \leftarrow S \cup \{u\} \)
7. \hspace{1em} \textbf{for each vertex} \( v \in \text{Adj}[u] \)
8. \hspace{2em} \textbf{do} \( \text{RELAX}(u, v, w) \)

\textbf{Note:} relax requires updating of min values in \( Q \).
Figure 24.6  The execution of Dijkstra’s algorithm. The source $s$ is the leftmost vertex. The shortest-path estimates are shown within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set $S$, and white vertices are in the min-priority queue $Q = V - S$. (a) The situation just before the first iteration of the while loop of lines 4–8. The shaded vertex has the minimum $d$ value and is chosen as vertex $u$ in line 5. (b)–(f) The situation after each successive iteration of the while loop. The shaded vertex in each part is chosen as vertex $u$ in line 5 of the next iteration. The $d$ and $\pi$ values shown in part (f) are the final values.
Analysis of Dijkstra’s Algorithm

• Correctness: Theorem 24.6 (Loop invariant)
• Min-priority queue operations
  – INSERT (line 3)
  – EXTRACT-MIN (line 5)
  – DECREASE-KEY (line 8)
• Time analysis
  – Line 4-8: while loop $\rightarrow O(V)$
  – Line 7-8: for loop and relaxation $\rightarrow |E|$
  – Running time depends on how to implement min-priority queue
    • Simple array: $O(V^2+E) = O(V^2)$
    • Binary min-heap: $O((V+E)\lg V)$
    • Fibonacci min-heap: $O(V\lg V + E)$
Edsger Wybe Dijkstra was one of the most influential members of computing science's founding generation. Among the domains in which his scientific contributions are fundamental are:

- algorithm design
- programming languages
- program design
- operating systems
- distributed processing
- formal specification and verification
- design of mathematical arguments
Examples of shortest paths depending on start node
All pairs shortest paths

- Diameter of a graph (longest shortest path)
- Calculate the shortest path from each source
- Find the longest shortest path...

- Means to estimate/approximate it
Euclidean Networks

- In applications where networks model maps, our primary interest is often in finding the best route from one place to another. In this section, we examine a strategy for this problem: a fast algorithm for the source–sink shortest-path problem in Euclidean networks, which are networks whose vertices are points in the plane and whose edge weights are defined by the geometric distances between the points.

- These networks satisfy two important properties that do not necessarily hold for general edge weights. First, the **distances satisfy the triangle inequality**: The distance from \( s \) to \( d \) is never greater than the distance from \( s \) to \( x \) plus the distance from \( x \) to \( d \). Second, **vertex positions give a lower bound on path length**: No path from \( s \) to \( d \) will be shorter than the distance from \( s \) to \( d \). The algorithm for the source–sink shortest-paths problem that we examine in this section takes advantage of these two properties to improve performance.
Calculating paths by matrix operations
Paths of length 2
Diagonal 1 = self-loop
Diagonal 0 or 1

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Graphs and adjacency matrices for different diagonal conditions.
Transitive closure

• Transitive closure of a digraph G is a graph G’ with same vertices, and edge between any u and v from G if there is a path from u to v in G.
Transitive closure

\[ G * G * G * ... \]

\[ G[i][j] \text{ and } G[j][k] \Rightarrow G[i][k] \]

Exists link via j

1 on diagonal - link to itself

Figure 19.13
Transitive closure

This digraph (top) has just eight directed edges, but its transitive closure (bottom) shows that there are directed paths connecting 19 of the 30 pairs of vertices. Structural properties of the digraph are reflected in the transitive closure.

For example, rows 0, 1, and 2 in the adjacency matrix for the transitive closure are identical (as are columns 0, 1, and 2) because those vertices are on a directed cycle in the digraph.
Property 19.6 We can compute the transitive closure of a digraph by constructing the latter's adjacency matrix $A$, adding self-loops for every vertex, and computing $A^V$.

Proof: Continuing the argument in the previous paragraph, $A^3$ has an edge for every path of length less than or equal to 3 in the digraph, $A^4$ has an edge for every path of length less than or equal to 4 in the digraph, and so forth. We do not need to consider paths of length greater than $V$ because of the pigeonhole principle: Any such path must revisit some vertex (since there are only $V$ of them) and therefore adds no information to the transitive closure because the same two vertices are connected by a directed path of length less than $V$ (which we could obtain by removing the cycle to the revisited vertex).
Book: Sedgewick, Algorithms ...

- **19.3. Reachability and Transitive Closure**
The textbook algorithm for computing the product of two $V$-by-$V$ matrices computes, for each $s$ and $t$, the dot product of row $s$ in the first matrix and row $t$ in the second matrix, as follows:

```plaintext
for (s = 0; s < V; s++)
  for (t = 0; t < V; t++)
    for (i = 0, C[s][t] = 0; i < V; i++)
      C[s][t] += A[s][i]*B[i][t];
```

In matrix notation, we write this operation simply as $C = A \times B$. This operation is defined for matrices comprising any type of entry for which $0$, $+$, and $\times$ are defined. In particular, if the matrix entries are either true or false and we interpret $a+b$ to be the logical or operation and $a*b$ to be the logical and operation, then we have Boolean matrix multiplication. In Java, we can use the following version:

```plaintext
for (s = 0; s < V; s++)
  for (t = 0; t < V; t++)
    for (i = 0, C[s][t] = false; i < V; i++)
      if (A[s][i] && B[i][t]) C[s][t] = true;
```
Complexity...

• for \( i = 1 \) to \(|V|\) do \( V^{(i)} = V^{(i-1) \times V} \)
• \( V^3 \) operations for \( V^2, V^3, \ldots V^v \)
• \( \Rightarrow O(V^4) \)

• Use exponential: \( 2 \Rightarrow 4 \Rightarrow 8 \Rightarrow 16 \ldots \) steps.
• \( V^2 \times V^2 = V^4, V^4 \times V^4 = V^8, \ldots \Rightarrow O( \lceil \log V \rceil \times V^3 ) \)
• Can we avoid so many cycles?
closure with just one operation of this kind, building up the transitive closure from the adjacency matrix in place, as follows:

```c
for (i = 0; i < V; i++)
    for (s = 0; s < V; s++)
        for (t = 0; t < V; t++)
            if (A[s][i] && A[i][t]) A[s][t] = true;
```

This classical method, invented by S. Warshall in 1962, is the method of choice for computing the transitive closure of dense digraphs. The code is similar to the code that we might try to use to square a Boolean matrix in place: The difference (which is significant!) lies in the order of the for loops.

**Property 19.7** With Warshall's algorithm, we can compute the transitive closure of a digraph in time proportional to $V^3$. 
for (i = 0; i < V; i++)
    for (s = 0; s < V; s++)
        for (t = 0; t < V; t++)
            if (A[s][i] && A[i][t]) A[s][t] = true;

Paths via 0
Paths via 1 (including 0-1, 1-0)
...
Property 19.7  With Marshall's algorithm, we can compute the transitive closure of a digraph in time proportional to $V^3$.

Proof: The running time is immediately evident from the structure of the code. We prove that it computes the transitive closure by induction on $i$. After the first iteration of the loop, the matrix has true in row $s$ and column $t$ if and only if the digraph has either the edge $s-t$ or the path $s-0-t$. The second iteration checks all the paths between $s$ and $t$ that include 1 and perhaps 0, such as $s-1-t$, $s-1-0-t$, and $s-0-1-t$. We are led to the following inductive hypothesis: The $i$th iteration of the loop sets the bit in row $s$ and column $t$ in the matrix to true if and only if there is a directed path from $s$ to $t$ in the digraph that does not include any vertices with indices greater than $i$ (except possibly the endpoints $s$ and $t$). As just argued, the condition is true when $i$ is 0, after the first iteration of the loop. Assuming that it is true for the $i$th iteration of the loop, there is a path from $s$ to $t$ that does not include any vertices with indices greater than $i+1$ if and only if (i) there is a path from $s$ to $t$ that does not include any vertices with indices greater than $i$, in which case $A[s][t]$ was set on a previous iteration of the loop (by the inductive hypothesis); or (ii) there is a path from $s$ to $i+1$ and a path from $i+1$ to $t$, neither of which includes any vertices with indices greater than $i$ (except endpoints), in which case $A[s][i+1]$ and $A[i+1][t]$ were previously set to true (by hypothesis), so the inner loop sets $A[s][t]$. ■
Proof

- Proof: transitive closure by induction on i.
- Iteration 1: either s-t or the path s-0-t.
- Iteration 2: all the paths between s and t that include 1 and perhaps 0, such as s-1-t, s-1-0-t, and s-0-1-t.
- Inductive hypothesis: The ith iteration of the loop sets the bit (s, t) to true iff there is a directed path from s to t in the digraph that does not include any vertices with indices greater than i (except possibly the endpoints s and t).
• Assuming that it is true for the ith iteration of the loop, there is a path from s to t that does not include any vertices with indices greater than i+1 iff
  – (i) there is a path from s to t without indices >i, in which case A[s][t] was set on a previous iteration of the loop (inductive hypothesis)
  – (ii) there is a path from s to i+1 and a path from i+1 to t, neither of which includes any vertices with indices greater than i (except endpoints), in which case A[s][i+1] and A[i+1][t] were previously set to true (by hypothesis), so the inner loop sets A[s][t].
• How to further improve?

• Test for A[s][i] ...

•
Program 19.3 Warshall's algorithm

The constructor for class GraphTC computes the transitive closure of G in the private data field T so that clients can use GraphTC objects to test whether any given vertex in a digraph is reachable from any other given vertex. The constructor initializes T with a copy of G, adds self-loops, then uses Warshall's algorithm to complete the computation. We use a DenseGraph object for the transitive closure T because the algorithm needs an efficient implementation of the edge existence test (see Section 17.5).

class GraphTC
{
    private DenseGraph T;

    GraphTC(Graph G)
    {
        T = GraphUtilities.densecopy(G);
        for (int s = 0; s < T.V(); s++)
            T.insert(new Edge(s, s));
        for (int i = 0; i < T.V(); i++)
            for (int s = 0; s < T.V(); s++)
                if (T.edge(s, i))
                    for (int t = 0; t < T.V(); t++)
                        if (T.edge(i, t))
                            T.insert(new Edge(s, t));
    }

    boolean reachable(int s, int t)
    { return T.edge(s, t); }
}
Table 19.1 Empirical study of transitive-closure algorithms

This table shows running times that exhibit dramatic performance differences for various algorithms for computing the transitive closure of random digraphs, both dense and sparse. For all but the adjacency-lists DFS, the running time goes up by a factor of 8 when we double $V$, which supports the conclusion that it is essentially proportional to $V^3$. The adjacency-lists DFS takes time proportional to $VE$, which explains the running time roughly increasing by a factor of 4 when we double both $V$ and $E$ (sparse graphs) and by a factor of about 2 when we double $E$ (dense graphs), except that list-traversal overhead degrades performance for high-density graphs.

<table>
<thead>
<tr>
<th>sparse (10$V$ edges)</th>
<th>dense (250 vertices)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V$</td>
<td>$W$</td>
</tr>
<tr>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>50</td>
<td>3</td>
</tr>
<tr>
<td>125</td>
<td>35</td>
</tr>
<tr>
<td>250</td>
<td>275</td>
</tr>
<tr>
<td>500</td>
<td>2222</td>
</tr>
</tbody>
</table>

Key:
- $W$: Warshall's algorithm (Section 19.3)
- $W^*$: Improved Warshall's algorithm (Program 19.3)
- $A$: DFS, adjacency-matrix representation (Programs 19.4 and 17.7)
- $L$: DFS, adjacency-lists representation (Program 19.417.9)
Random walks...

Graph:
- A > B 0.95
- A > C 0.05
- B > D 0.7
- B > E 0.3
- C > E 1.0
- D > A 1.0
- E > D 0.2
- E > A 0.8
Matrix:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0.95</th>
<th>0.05</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.7</td>
<td>0.3</td>
</tr>
<tr>
<td>3.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.0</td>
</tr>
<tr>
<td>4.</td>
<td>1.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5.</td>
<td>0.8</td>
<td>0</td>
<td>0</td>
<td>0.2</td>
<td>0</td>
</tr>
</tbody>
</table>
Matrix:

\[
\begin{pmatrix}
0 & 0.95 & 0.05 & 0 & 0 \\
0 & 0 & 0 & 0.7 & 0.3 \\
0 & 0 & 0 & 0 & 1.0 \\
1.0 & 0 & 0 & 0 & 0 \\
0.8 & 0 & 0 & 0.2 & 0 \\
\end{pmatrix}
\]

Random Walk with 100000 steps

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.32589</td>
<td>0.30974</td>
<td>0.01614</td>
<td>0.23799</td>
<td>0.11024</td>
</tr>
</tbody>
</table>

Matrix multiplications with 10000 steps

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.326051516139543</td>
<td>0.309748940332566</td>
<td>0.0163025758069772</td>
<td>0.238669709814146</td>
<td>0.10922;</td>
</tr>
<tr>
<td>0.326051516139543</td>
<td>0.309748940332566</td>
<td>0.0163025758069771</td>
<td>0.238669709814145</td>
<td>0.10922;</td>
</tr>
<tr>
<td>0.326051516139543</td>
<td>0.309748940332566</td>
<td>0.0163025758069772</td>
<td>0.238669709814146</td>
<td>0.10922;</td>
</tr>
<tr>
<td>0.326051516139543</td>
<td>0.309748940332566</td>
<td>0.0163025758069772</td>
<td>0.238669709814146</td>
<td>0.10922;</td>
</tr>
<tr>
<td>0.326051516139543</td>
<td>0.309748940332566</td>
<td>0.0163025758069772</td>
<td>0.238669709814146</td>
<td>0.10922;</td>
</tr>
</tbody>
</table>
Finding the modules

Public datasets for H.sapiens
- IntAct: Protein interactions (PPI), 18773 interactions
- IntAct: PPI via orthologs from IntAct, 6705 interactions
- MEM: gene expression similarity over 89 tumor datasets, 46286 interactions
- Transfac: gene regulation data, 5183 interactions
Finding the modules

Public datasets for *H. sapiens*

- Red: IntAct: Protein interactions (PPI), 18773 interactions
- Green: IntAct: PPI via orthologs from IntAct, 6705 interactions
- Blue: MEM: gene expression similarity over 89 tumor datasets, 46286 interactions
- Light blue: Transfac: gene regulation data, 5183 interactions
Module evaluation

GO: Transforming growth factor beta signaling pathway, embryonic development, gastrulation
KEGG: Cell cycle, cancers, WNT pathway

GO: JAK-STAT cascade, Kinase inhibitor activity, Insulin receptor signaling pathway
KEGG: Type II diabetes mellitus

GO: Brain development, Pigment granule, Melanine metabolic process

Jüri Reimand: GraphWeb.
Genome Informatics, CSHL. Nov 1 2007
MCL clustering algorithm

- Markov (Chain Monte Carlo) Clustering
  - [http://www.micans.org/mcl/](http://www.micans.org/mcl/)

- Random walks according to edge weights

- Follow the different paths according to their probability

- Regions that are traversed “often” form clusters
http://www.micans.org/mcl/intro.html

With this, the MCL algorithm can be written as

\[
\begin{align*}
G & \text{ is a graph} \\
\text{add loops to } G & \quad \# \text{ see below} \\
\text{set } \Gamma & \text{ to some value} \quad \# \text{ affects granularity} \\
\text{set } M_1 & \text{ to be the matrix of random walks on } G
\end{align*}
\]

while (change) {
    \[
    \begin{align*}
    M_2 & = M_1 \times M_1 \quad \# \text{ expansion} \\
    M_1 & = \Gamma(M_2) \quad \# \text{ inflation} \\
    \text{change} & = \text{difference}(M_1, M_2)
    \end{align*}
    \]
}

set CLUSTERING as the components of \( M_1 \) \quad \# \text{ see below}
MAXIMUM FLOW

Max-Flow Min-Cut Theorem (Ford Fukerson’s Algorithm)
What is Network Flow?

Flow network is a directed graph $G=(V,E)$ such that each edge has a non-negative capacity $c(u,v) \geq 0$.

Two distinguished vertices exist in $G$ namely:

- **Source** (denoted by $s$): In-degree of this vertex is 0.
- **Sink** (denoted by $t$): Out-degree of this vertex is 0.

Flow in a network is an integer-valued function $f$ defined on the edges of $G$ satisfying $0 \leq f(u,v) \leq c(u,v)$, for every edge $(u,v)$ in $E$. 
What is Network Flow?

• Each edge \((u,v)\) has a non-negative capacity \(c(u,v)\).

• If \((u,v)\) is not in \(E\) assume \(c(u,v)=0\).

• We have source \(s\) and sink \(t\).

• Assume that every vertex \(v\) in \(V\) is on some path from \(s\) to \(t\).

Following is an illustration of a network flow:

![Network Flow Diagram](image)

- \(c(s,v1)=16\)
- \(c(v1,s)=0\)
- \(c(v2,s)=0\) ...
Conditions for Network Flow

For each edge \((u,v)\) in \(E\), the flow \(f(u,v)\) is a real valued function that must satisfy following 3 conditions:

- **Capacity Constraint**: \(\forall \ u, v \in V, \ f(u,v) \leq c(u,v)\) (flow < capacity)
- **Skew Symmetry**: \(\forall \ u, v \in V, \ f(u,v) = -f(v,u)\) (inflow = -outflow)
- **Flow Conservation**: \(\forall \ u \in V - \{s,t\} \ \sum_{v \in V} f(u,v) = 0\) (net flow = 0)

Skew symmetry condition implies that \(f(u,u) = 0\).
The Value of a Flow.

The value of a flow is given by:

\[ |f| = \sum_{v \in V} f(s, v) = \sum_{v \in V} f(v, t) \]

The flow into the node is same as flow going out from the node and thus the flow is conserved. Also the total amount of flow from source \( s \) = total amount of flow into the sink \( t \).
Example of a flow

Table illustrating Flows and Capacity across different edges of graph above:

- $f_{s,1} = 9$, $c_{s,1} = 10$ (Valid flow since $10 > 9$)
- $f_{s,2} = 6$, $c_{s,2} = 6$ (Valid flow since $6 \geq 6$)
- $f_{1,2} = 1$, $c_{1,2} = 1$ (Valid flow since $1 \geq 1$)
- $f_{1,t} = 8$, $c_{1,t} = 8$ (Valid flow since $8 \geq 8$)
- $f_{2,t} = 7$, $c_{2,t} = 10$ (Valid flow since $10 > 7$)

The flow across nodes 1 and 2 are also conserved as flow into them = flow out.
The Maximum Flow Problem

Given a Graph G (V,E) such that:

- $x_{i,j} = \text{flow on edge } (i,j)$
- $u_{i,j} = \text{capacity of edge } (i,j)$
- $s = \text{source node}$
- $t = \text{sink node}$

Maximize $v$

Subject To

- $\sum_j x_{ij} - \sum_j x_{ji} = 0 \text{ for each } i \neq s,t$
- $\sum_j x_{sj} = v$
- $0 \leq x_{ij} \leq u_{ij} \text{ for all } (i,j) \in E.$

In simple terms maximize the s to t flow, while ensuring that the flow is feasible.
A Cut in a network is a partition of $V$ into $S$ and $T$ ($T=V-S$) such that $s$ (source) is in $S$ and $t$ (target) is in $T$. 

![Graph with labeled nodes and edges showing a cut](image_url)
Capacity of Cut \((S, T)\)

\[
c(S, T) = \sum_{u \in S, v \in T} c(u, v)
\]
Min s-t cut (Also called as a Min Cut) is a cut of minimum capacity.
Flow of Min Cut (Weak Duality)

Let $f$ be the flow and let $(S, T)$ be a cut. Then $|f| \leq \text{CAP}(S, T)$.

In maximum flow, minimum cut problems forward edges are full or saturated and the backward edges are empty because of the maximum flow. Thus maximum flow is equal to capacity of cut. This is referred to as weak duality.

Proof:

$$|f| = \left\{ \begin{array}{l} \sum_{e \text{ out of } S} f(e) - \sum_{e \text{ in to } S} f(e) \\ 0 \end{array} \right.$$  

$$= \sum_{e \text{ out of } S} f(e)$$

$$= \sum_{e \text{ out of } S} u(e)$$

$$= \text{cap}(S, T)$$
Methods

Max-Flow Min-Cut Theorem

• The Ford-Fulkerson Method

• The Preflow-Push Method
The Ford-Fulkerson Method

- Try to improve the flow, until we reach the maximum value of the flow

- The residual capacity of the network with a flow $f$ is given by:

  The residual capacity $(rc)$ of an edge $(i,j)$ equals $c(i,j) - f(i,j)$ when $(i,j)$ is a forward edge, and equals $f(i,j)$ when $(i,j)$ is a backward edge. Moreover the residual capacity of an edge is always non-negative.

$$c_f(u, v) = c(u, v) - f(u, v)$$

Original Network

Residual Network
The Ford-Fulkerson Method

Begin
x := 0; // x is the flow.
create the residual network G(x);
while there is some directed path from s to t in G(x) do begin
let P be a path from s to t in G(x);
Δ := δ(P);
send Δ units of flow along P;
update the r's;
end
end {the flow x is now maximum}.
Augmenting Paths (A Useful Concept)

Definition:

An **augmenting path** $p$ is a simple path from $s$ to $t$ on a residual network that is an alternating sequence of vertices and edges of the form $s, e_1, v_1, e_2, v_2, ..., e_k, t$ in which no vertex is repeated and no forward edge is saturated and no backward edge is free.

Characteristics of augmenting paths:

- We can put more flow from $s$ to $t$ through $p$.
- The edges of residual network are the edges on which residual capacity is positive.
- We call the maximum capacity by which we can increase the flow on $p$ the **residual capacity** of $p$.

$$c_f(p) = \min \{c_f(u, v) : (u, v) \text{ is on } p\}$$
The Ford-Fulkerson’s Algorithm

**FORDFULKERSON** (*G*, *E*, *s*, *t*)

**FOREACH** *e* ∈ *E*

\[ f(e) \leftarrow 0 \]

\[ G_f \leftarrow \text{residual graph} \]

**WHILE** (there exists augmenting path *P*)

\[ f \leftarrow \text{augment}(f, P) \]

update \( G_f \)

**ENDWHILE**

**RETURN** \( f \)

**AUGMENT** (*f*, *P*)

\[ b \leftarrow \text{bottleneck}(P) \]

**FOREACH** *e* ∈ *P*

**IF** (*e* ∈ *E*)

// backwards arc

\[ f(e) \leftarrow f(e) + b \]

**ELSE**

// forward arc

\[ f(e^R) \leftarrow f(e) - b \]

**RETURN** \( f \)
Proof of correctness of the algorithm

**Lemma:** At each iteration all residual capacities are integers.

**Proof:** It’s true at the beginning. Assume it’s true after the first k-1 augmentations, and consider augmentation k along path P. The residual capacity $\Delta$ of P is the smallest residual capacity on P, which is integral.

After updating, we modify the residual capacities by 0 or $\Delta$, and thus residual capacities stay integers.

**Theorem:** Ford-Fulkerson’s algorithm is finite

**Proof:** The capacity of each augmenting path is at least 1. The augmentation reduces the residual capacity of some edge (s,j) and doesn’t increase the residual capacity for some edge (s,i) for any i.

So the sum of residual capacities of edges out of s keeps decreasing, and is bounded below 0.

Number of augmentations is $O(nC)$ where C is the largest of the capacity in the network.
When is the flow optimal?

A flow $f$ is maximum flow in $G$ if:

1. The residual network $G_f$ contains no more augmented paths.
2. $|f| = c(S,T)$ for some cut $(S,T)$ (a min-cut)

Proof:

1. Suppose there is an augmenting path in $G_f$ then it implies that the flow $f$ is not maximum, because there is a path through which more data can flow. Thus if flow $f$ is maximum then residual n/w $G_f$ will have no more augmented paths.

2. Let $v = Fx(S,T)$ be the flow from $s$ to $t$. By assumption $v = \text{CAP}(S,T)$. By Weak duality, the maximum flow is at most $\text{CAP}(S,T)$. Thus the flow is maximum.
The Ford-Fulkerson Augmenting Path Algorithm for the Maximum Flow Problem

15.082 and 6.855J  (MIT OCW)
This is the original network, and the original residual network.
Ford-Fulkerson Max Flow

Find any s-t path in $G(x)$
Ford-Fulkerson Max Flow

Determine the capacity $\Delta$ of the path.

Send $\Delta$ units of flow in the path. Update residual capacities.
Ford-Fulkerson Max Flow

Find any s-t path
Ford-Fulkerson Max Flow

Determine the capacity $\Delta$ of the path.

Send $\Delta$ units of flow in the path.
Update residual capacities.
Ford-Fulkerson Max Flow

Find any s-t path
Determine the capacity $\Delta$ of the path.

Send $\Delta$ units of flow in the path.

Update residual capacities.
Ford-Fulkerson Max Flow

Find any s-t path
Ford-Fulkerson Max Flow

Determine the capacity $\Delta$ of the path.

Send $\Delta$ units of flow in the path.
Update residual capacities.
Ford-Fulkerson Max Flow

Find any s-t path
Determine the capacity $\Delta$ of the path.

Send $\Delta$ units of flow in the path.

Update residual capacities.
There is no s-t path in the residual network. This flow is optimal.
Ford-Fulkerson Max Flow

These are the nodes that are reachable from node s.
Here is the optimal flow
Counterexample for termination

Ülesanne 54. Vaatleme voogu järgmisel joonisel.

Olgu $R = \frac{\sqrt{5}-1}{2}$ (siis $R^n = R^{n+1} + R^{n+2}$) ja ülejäänud servadel suured lähilaskevõimed. Olgu esimene suurendav ahel $s - a - d - t$ ning järgmised suurendavad ahelad (tsüklis)

1. $s - c - f - d - a - b - e - t$
2. $s - b - e - f - c - a - d - t$
3. $s - a - d - e - b - c - f - t$.

Näita, et Ford-Fulkersoni algoritm ei lõpeta tööd.
Distribution & Transportation

<table>
<thead>
<tr>
<th>supply</th>
<th>channels</th>
<th>cost</th>
<th>cap</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: 4</td>
<td>0-3:</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1: 4</td>
<td>0-7:</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2: 6</td>
<td>1-4:</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>distribution</td>
<td>2-4:</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2-9:</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>3-5:</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>3-6:</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>4-5:</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>demand</td>
<td>4-6:</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>7: 7</td>
<td>5-7:</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>8: 3</td>
<td>5-8:</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>9: 4</td>
<td>6-9:</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>supply</th>
<th>channels</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: 3</td>
<td>0-6:</td>
<td>2</td>
</tr>
<tr>
<td>1: 4</td>
<td>0-7:</td>
<td>1</td>
</tr>
<tr>
<td>2: 6</td>
<td>0-8:</td>
<td>5</td>
</tr>
<tr>
<td>3: 3</td>
<td>1-6:</td>
<td>3</td>
</tr>
<tr>
<td>4: 2</td>
<td>1-5:</td>
<td>1</td>
</tr>
<tr>
<td>demand</td>
<td>2-8:</td>
<td>3</td>
</tr>
<tr>
<td>5: 6</td>
<td>2-9:</td>
<td>4</td>
</tr>
<tr>
<td>6: 6</td>
<td>3-6:</td>
<td>2</td>
</tr>
<tr>
<td>7: 7</td>
<td>3-7:</td>
<td>1</td>
</tr>
<tr>
<td>8: 3</td>
<td>4-9:</td>
<td>6</td>
</tr>
<tr>
<td>9: 4</td>
<td>4-5:</td>
<td>3</td>
</tr>
</tbody>
</table>
Job placement:
6 people, 6 jobs, preferences...

<table>
<thead>
<tr>
<th>Alice</th>
<th>Adobe</th>
<th>Adobe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apple</td>
<td>HP</td>
<td>Alice</td>
</tr>
<tr>
<td>HP</td>
<td>Bob</td>
<td>Bob</td>
</tr>
<tr>
<td>Bob</td>
<td>Apple</td>
<td>Alice</td>
</tr>
<tr>
<td>Adobe</td>
<td>Apple</td>
<td>Bob</td>
</tr>
<tr>
<td>Apple</td>
<td>Dave</td>
<td>Dave</td>
</tr>
<tr>
<td>Yahoo</td>
<td>Dave</td>
<td>Dave</td>
</tr>
<tr>
<td>Carol</td>
<td>HP</td>
<td>Alice</td>
</tr>
<tr>
<td>HP</td>
<td>IBM</td>
<td>Carol</td>
</tr>
<tr>
<td>IBM</td>
<td>Sun</td>
<td>Frank</td>
</tr>
<tr>
<td>Sun</td>
<td>IBM</td>
<td>Carol</td>
</tr>
<tr>
<td>IBM</td>
<td>Carol</td>
<td>Frank</td>
</tr>
<tr>
<td>Carol</td>
<td>Eliza</td>
<td>Eliza</td>
</tr>
<tr>
<td>Eliza</td>
<td>Sun</td>
<td>Eliza</td>
</tr>
<tr>
<td>Sun</td>
<td>Yahoo</td>
<td>Frank</td>
</tr>
<tr>
<td>Yahoo</td>
<td>Bob</td>
<td>Bob</td>
</tr>
<tr>
<td>Bob</td>
<td>Eliza</td>
<td>Eliza</td>
</tr>
<tr>
<td>Eliza</td>
<td>Frank</td>
<td>Frank</td>
</tr>
</tbody>
</table>
Converting the Matching problem to Network Flow
Converting Matching to Network Flow
Converting Optimal Bipartite Matching to Network Flow