WEIGHTED GRAPH ALGORITHMS

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

\[
    w(T) = \sum_{uv \in T} w(u, v)
\]

- May have more than one MST with the same weight

- Two classic algorithms: \(O(E \log V)\) 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.
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.

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 some MST
  - \((u, v)\) is called a safe edge for \( A \)

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 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

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?
1.4.2011

Illustration of Theorem 23.1

- \( A = \{(a, b), (c, d), (g, h)\} \)
- \( S = \{a, b, c, d, e\}; V - S = \{g, h, f, d\} \)
- \( \) 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 \( W(T') = W(T) - w(x, y) + w(u, v) = W(T) \)
- \((u, v)\) is actually a safe edge for \( A \):
  - Since \( A \subseteq T \) and \((x, y) \notin A \implies A \cup \{(u, v)\} \subseteq T' \)
- \( A \cup \{(u, v)\} \subseteq T' \)

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 components of \( G_A \) is a tree
- Any safe edge \((u, v)\) for \( A \) connects distinct components 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 form 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\(e=(V, A)\)
- 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 r to start the tree.
- 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
This creates a spanning tree, since no cycle can be introduced, but is it minimum?

Prim’s Algorithm in Action

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
- $\pi[v]$ = parent of $v$ in the tree
- $A = \{(v, \pi[v]) : v \in V \setminus \{r\} \}$: finally $Q = \emptyset$

**Illustration of MST-PRIM**

Properties of MST-PRIM

- Prior to each iteration of the while loop of lines 6—11
  - $A = \{(v, \pi[v]) : v \in V \setminus \{r\} \}$
    - The vertices already placed into the MST are those in $V \setminus Q$.
    - For all vertices $v \in Q$, if $\pi[v] = \text{NIL}$, then $\text{key}[v] = \infty$ and key $[v]$ is the weight of a light edge $(v, \pi[v])$ connecting $v$ to some vertex already placed into the MST.
- Line 7: Identify a vertex $u \in Q$ incident on a light edge crossing $(V \setminus Q, Q)$.
- Lines 8—11: Update key and $\pi$ 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 $Q$.
  - $\text{BUILD-MIN-HEAP}$ (line 5): $O(V)$
    - The body of while loop is executed $|V|$ times.
      - $\text{EXTRACT-MIN}$: $O(\lg V)$
      - The for loop in lines 8—11 is executed $O(E)$ times altogether.
      - Line 11: $\text{DECREASE-KEY}$ operation: $O(\lg V)$
    - Total performance = $O(V \lg V + E \lg V)$.
- Use Fibonacci heap to implement the min-priority queue $Q$.
  - $O(E + V \lg V)$

**MST-PRIM**

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

$\text{key}[h]=8; \text{key}[c]=8; \text{key}[i]=2; \text{key}[f]=4$

$\pi[u]=c, \text{adj}[c]=\{b, i, d, f\}$

$\pi[u]=b, \text{adj}[b]=\{a, c, h\}$

$\pi[u]=a, \text{adj}[a]=\{b, h\}$

$\pi[d]=a, \text{adj}[d]=\{a\}$

$\pi[h]=a, \text{adj}[h]=\{b, a, c\}$

$\text{key}[h]=8; \text{key}[c]=8; \text{key}[i]=2; \text{key}[f]=4$
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 \((x, y)\) 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 \(O(m \log m)\) 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!
- \(\text{Same component}(v_1, v_2)\) – Do vertices \(v_1\) and \(v_2\) lie in the same connected component of the current graph?
- \(\text{Merge components}(C_1, C_2)\) – Merge the given pair of connected components into one component.
Fast Kruskal Implementation

Put the edges in a heap

\[ \text{count} = 0 \]

while (count < \( n - 1 \)) do

get next edge \((v, w)\)

if (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 \( O(m \log n) \).

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

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

- **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.

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

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 \( O(\alpha(n)) \) time, where \( \alpha(n) \) is the inverse Ackermann function and \( \alpha(n) \) (number of atoms in the universe) = 5.

<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 (FBS, heap)</td>
<td>( E \log V )</td>
<td>linear unless extremely sparse</td>
</tr>
<tr>
<td>Kruskal</td>
<td>( E \log E )</td>
<td>sort cost dominates</td>
</tr>
<tr>
<td>Kruskal (partial sort)</td>
<td>( E + X \log V )</td>
<td>cost depends on longest edge</td>
</tr>
<tr>
<td>Boruvka</td>
<td>( X \log V )</td>
<td>conservative upper bound</td>
</tr>
</tbody>
</table>
SINGLE-SOURCE SHORTEST PATHS (CHAPTER 24)

Example: Predictive Mobile text Entry Messaging...
What was the message?

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.

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 a path $p=v_0v_1...v_k$ is the sum of the weights of its constituent edges:

  $$w(p) = \sum_{e \in p} w(e)$$

- We define the shortest-path weight from $u$ to $v$ by

  $$d(u,v) = \min \{ w(p): u \rightarrow v \}$$

  If there is a path from $u$ to $v$, $d(u,v) \neq \infty$; otherwise, $d(u,v) = \infty$.

- A shortest path from vertex $u$ to vertex $v$ is then defined as any path with $w(p)=d(u,v)$.

**Variants**

- Single-source shortest paths problem – greedy
- 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 is 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.

**Optimal Substructure of a Shortest-Path**

**Lemma 24.1** (Subpath of shortest paths are shortest paths). Let $p=v_0v_1...v_k$ be a shortest path from vertex $v_0$ to $v_k$. Then for any $i$ and $j$ such that $1 \leq i \leq j \leq k$, let $p_{ij} = v_{i0}v_{i1}...v_{ij}$ 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.

```
for each vertex $v \in V(G)$
    do $d[v] \leftarrow \infty$
    $\pi[v] \leftarrow$ NIL

$d[s] \leftarrow 0$
```

**Predecessor of $v$ in the shortest path**
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 \(\pi[v]\).

\[
\begin{align*}
1. & \text{ RELAX}(u, v, w) \\
2. & \text{ for each edge } (u, v) \in G.E \\
3. & \text{ if } v.d > u.d + w(u, v) \\
4. & \text{ then } d[v] \leftarrow d[u] + w(u, v) \\
5. & \text{ and } \pi[v] \leftarrow u \\
6. & \text{ return } \text{FALSE} \\
7. & \text{ return } \text{TRUE}
\end{align*}
\]

Bellman-Ford

Bellman-Ford \((G, w, s)\)

1. \text{ Initialise-Single-Source}(G,S)
2. \text{ for } i=1 \text{ to } |G.V|-1
3. \text{ for each edge } (u,v) \in G.E
4. \text{ RELAX}(u, v, w)
5. \text{ for each edge } (u,v) \in G.E
6. \text{ if } v.d > u.d + w(u,v)
7. \text{ return } \text{FALSE}
8. \text{ return } \text{TRUE}

Shortest paths on a DAG

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

\(O(V + E)\)
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 $s$ to $S$, and relaxes all edges leaving $u$

Dijkstra’s Algorithm (Cont.)

\begin{algorithm}
\begin{algorithmic}
  \STATE Initialize-Single-Source($G, s$)
  \STATE $S \leftarrow \emptyset$
  \STATE $Q \leftarrow V[G]$
  \WHILE{$Q \neq \emptyset$}
    \STATE $u \leftarrow \text{EXTRACT-MIN}(Q)$
    \STATE $S \leftarrow S \cup \{u\}$
    \FOR {each vertex $v \in Adj[u]$}
      \STATE $\text{RELAX}(u, v, w)$
    \ENDFOR
  \ENDWHILE
\end{algorithmic}
\end{algorithm}

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: $O(V + E)$
    - Fibonacci heap: $O(E + \log V)$
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

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

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.
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:

\[
\text{for } i = 1 \text{ to } |V| \text{ do}
\begin{align*}
V(i) &= V(i-1) + V(i) \\
\end{align*}
\]

In matrix notation, we write this operation simply as $C = A + B$.

This operation is defined for matrices comprising any type of entry for which $+$ and $*$ are defined. In particular, if the matrix entries are either true or false and we interpret $+$ to be the logical or operation and $*$ to be the logical and operation, then we have boolean matrix multiplication. In Java, we can use the following version:

\[
\text{for } (s = 0; s < V; s++) \\
\text{for } (t = 0; t < V; t++)
\begin{align*}
&\text{if } (A[i][t] \&\& B[i][t]) \text{ then } C[i][t] = true; \\
&\text{else } C[i][t] = false;
\end{align*}
\]

Complexity...

- for $i=1$ to $|V|$ do $V^{(i+1)} = V^{(i)} + V$
- $V^3$ operations for $V^2$, $V^3$, ... $V^v$
- $\Rightarrow O(V^4)$

- Use exponential: $2 \Rightarrow 4 \Rightarrow 8 \Rightarrow 16 \ldots$ steps.
- $V^2 * V^2 = V^4$, $V^4 * V^4 = V^8$, ... $\Rightarrow O(\log V \times V^3)$
- Can we avoid so many cycles?

Book: Sedgewick, Algorithms ...

- 19.3. Reachability and Transitive Closure

Diagonal 0 or 1

Property 19.4. We can compute the transitive closure of a digraph by constructing the digraph’s adjacency matrix $A$, adding self-loops for every vertex, and computing $A^k$.

Proof: Continuing the argument in the previous paragraph, $A^k$ has an edge for every path of length less than or equal to $k$ in the digraph. $A^0$ has an edge for every path of length less than or equal to 0 in the digraph, and so forth. We do not need to consider paths of length greater than $k$ because of the pigeonhole principle: two such paths must revisit some vertex (since there are only $V$ of them), and therefore such a path can be broken into a directed path of length less than $k$ which we could derive by removing the cycle in the iterated vertex.

Paths via 0
Paths via 1 (including 0-1, 1-0)
1.4.2011

**Property 19.7** With Warshall’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 $s\to t$ or the path $s\to \ldots \to t$. The second iteration checks all the paths between $s$ and $t$ that include 1 and perhaps 0, such as $s\to 1\to t$, $s\to 0\to t$, and $s\to 0\to 1\to t$.

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 $i = 0$, 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 $t$ without 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 $t$ 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 $s\to t$ or the path $s\to 0\to t$.
- **It 2**: all the paths between $s$ and $t$ that include 1 and perhaps 0, such as $s\to 1\to t$, $s\to 0\to t$, and $s\to 0\to 1\to t$.
- **Inductive hypothesis**: The $i$th 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 $t$).

**How to further improve?**

- Test for $A[s][i]$...
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:
1. 0 0.95 0.05 0 0
2. 0 0 0 0.7 0.3
3. 0 0 0 0 1.0
4. 1.0 0 0 0 0
5. 0.8 0 0 0.2 0
1.4.2011

**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**

---

### MAXIMUM FLOW

Max-Flow Min-Cut Theorem (Ford Fulkerson’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:
 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:

- **Skew Symmetry**: \(\forall u,v \in V, f(u,v) = -f(v,u)\) (inflow = -outflow)
- **Capacity Constraint**: \(\forall u,v \in V, f(u,v) \leq c(u,v)\) (flow < capacity)
- **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:

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

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

![Flow Network Diagram]

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

<table>
<thead>
<tr>
<th>Edge</th>
<th>Flow</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>(s,1)</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>(s,2)</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>(1,2)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(1,t)</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>(2,t)</td>
<td>7</td>
<td>10</td>
</tr>
</tbody>
</table>

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:

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

Maximize \(v\)

Subject To

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

In simple terms maximize \(v\), the s to t flow, while ensuring that the flow is feasible.

Cuts of Flow Networks

A Cut in a network is a partition of \(V\) into \(S\) and \(T = V - S\) such that \(s\) (source) is in \(S\) and \(t\) (target) is in \(T\).

![Cut Diagram]

Capacity of Cut \((S,T)\)

\[
c(S,T) = \sum_{u \in S, v \in T} c(u,v)
\]

Capacity = 30
Min Cut

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| ≤ 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.

Methods

Max-Flow Min-Cut Theorem

- The Ford-Fulkerson Method
- The Preflow-Push Method

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,e₁,v₁,e₂,v₂,...,eₖ,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.

\[ c_p^*(u,v) = \min \{ c_p(u,v) : (u,v) \text{ is on } p \} \]
The Ford-Fulkerson's Algorithm

\[
\begin{align*}
\text{FORDFULKERSON}(G, E, s, t) \\
\text{FOREACH } e \in E \\
f(e) \leftarrow 0 \\
G_f \leftarrow \text{residual graph} \\
\text{WHILE } (\text{there exists augmenting path } P) \\
f \leftarrow \text{augment}(f, P) \\
\text{update } G_f \\
\text{ENDWHILE} \\
\text{RETURN } f
\end{align*}
\]

The Ford-Fulkerson Augmenting Path Algorithm for the Maximum Flow Problem

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 \( k \)-th 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 in \( G \) if:

1. The residual network \( G_f \) contains no more augmented paths.
2. \(| f | = \text{CAP}(S, T) \) 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 \( G_f \) will have no more augmented paths.

2. Let \( v = \text{flow}(S, T) \) be the flow from \( s \) to \( t \). By assumption \( v \leq \text{CAP}(S, T) \).

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)

Ford-Fulkerson Max Flow

This is the original network, and the original residual network.

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

1. Determine the capacity $\Delta$ of the path.
2. Send $\Delta$ units of flow in the path.
3. Update residual capacities.

Find any s-t path
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.

These are the nodes that are reachable from node $s$.

Here is the optimal flow.
Counterexample for termination

Example 1.4. Vessenyi vagy jövővel jönnejű.

\[ R = \begin{bmatrix} a & b & c \end{bmatrix} \begin{bmatrix} d & e & f \end{bmatrix} = \begin{bmatrix} g \end{bmatrix} \]

Informal, this is a case where the trivial assignment is non-optimal. Let us see one more similar example:

1. \( a - e - f - d - a - b - c \)
2. \( a - b - c - f - c - a - d - e \)
3. \( a - b - d - e - c - f - b \)

Nézze meg, et Ford-Fulkerson algoritmus és lépésszámok.

Distribution & Transportation

Converting the Matching problem to Network Flow

Job placement: 6 people, 6 jobs, preferences...

Converting Optimal Bipartite Matching to Network Flow