1. **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**: \(O(E \log V)\) Greedy Algorithms
     - Kruskal's algorithm
     - Prim's algorithm

2. **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.

3. **The only MST of a planar graph**

4. **Equal weights in left fully connected 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.

Fully connected graph. Find a MST?

MST

MST-approximation of TSP
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 safe edge for $A$

**GENERIC-MST**

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

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

**Illustration of Theorem 23.1**

- $A = \{(a, b), (c, i), (h, g), (g, f)\}$
- $S = \{a, b, c, i, d\}$; $V - S = \{h, g, f, d\}$
- 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
  - $T' = T \cup \{(u, v)\}$
  - $T'$ is also a MST since $W(T') = W(T) + w(u, v) + w(u, v) \leq W(T)$
- $(u, v)$ is actually a safe edge for $A$
  - Since $A \subseteq T$ and $(u, v) \notin A \Rightarrow A \subseteq T'$
  - $A \cup \{(u, v)\} \subseteq T'$
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 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 component (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

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)**

1. Select an arbitrary vertex \( s \) to start the tree from.
2. 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

1. for each u ∈ V
   2. do D[u] ← ∞
   3. |D[r]| ← 0
   4. MH ← make-heap(D, V) // No edges
   5. T ← ∅
   6. while MH ≠ ∅ do
      7. do (u, e) ← MH.extractMin()
      8. add (u, e) to T
      9. for each v ∈ adjacent {u} do
         10. if v ∈ MH && w(u, v) < D[v]
            11. then D[v] ← w(u, v)
            12. MH.decreaseDistance(D[v], v, (u, v))
      13. return T // T is a MST

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 key[v], the minimum weight of any edge connecting v to a vertex in the tree
    – Key[v] = ∞ if no such edge
  • π[v] = parent of v in the tree
  • A = {(v, π[v]): v ∈ V−{r}−Q} ➔ finally Q = ∅
Illustration of MST-PRIM

Properties of MST-PRIM

 Prior to each iteration of the while loop of lines 6—11
- \( A = \{(v, \pi[v]) : v \in V-(r)-Q\} \)
- The vertices already placed into the MST are those in \( V-Q \)
- For all vertices \( v \in Q \), if \( \pi[v] \neq \text{NIL} \), then \( \text{key}[v] < \infty \) and \( \text{key}[v] \) 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 \( V-Q, Q \) \( \rightarrow \) add \( u \) to \( V-Q \) and \( (u, \pi[u]) \) to \( A \)
- 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 \( Q \)
  - BUILD-MIN-HEAP (line 5): \( O(V) \)
  - The body of while loop is executed \( |V| \) times
  - 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 \( (x, y) \) 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 is \( O(n^2) \), 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.

```
KRUSKAL(G):
1  A = ∅
2  foreach v ∈ G.V:
3    MAKE-SET(v)
4  foreach (u, v) ordered by weight(u, v), increasing:
5    if FIND-SET(u) ≠ FIND-SET(v):
6      A = A ∪ {(u, v)}
7      UNION(u, v)
8  return A
```

---

**Fast Kruskal Implementation**

Put the edges in a heap

\[
\text{count} = 0
\]

while \((\text{count} < n - 1)\) do

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

if \((\text{component}(v) ≠ \text{component}(w))\)

add to \(I\)

\(\text{component}(v) = \text{component}(w)\)

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 \(O(m \log m)\)?

---

**Kruskal algorithm for MST**

```
KRUSKAL(G):
1  A = ∅
2  foreach v ∈ G.V:
3    MAKE-SET(v)
4  foreach (u, v) ordered by weight(u, v), increasing:
5    if FIND-SET(u) ≠ FIND-SET(v):
6      A = A ∪ {(u, v)}
7      UNION(u, v)
8  return A
```

---

**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!

- **Some 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.
Union-Find Programs

We need a data structure for maintaining sets which can test if two elements are in the same set 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 *find(i)* now equals *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 *(u, v)* of weight *w*(uv) 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.

### Table 30.1: 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><em>V</em>^2</td>
<td>optimal for dense graphs</td>
</tr>
<tr>
<td>Prim (PES, heap)</td>
<td><em>E</em> log <em>V</em></td>
<td>conservative upper bound</td>
</tr>
<tr>
<td>Prim (PES, G-heap)</td>
<td><em>E</em> log <em>V</em></td>
<td>linear unless extremely sparse</td>
</tr>
<tr>
<td>Kruskal</td>
<td><em>E</em> log <em>V</em></td>
<td>cost dominates</td>
</tr>
<tr>
<td>Kruskal (partial sort)</td>
<td><em>E</em> log <em>V</em></td>
<td>cost depends on longest edge</td>
</tr>
<tr>
<td>Boruvka</td>
<td><em>E</em> log <em>V</em></td>
<td>conservative upper bound</td>
</tr>
</tbody>
</table>

Advanced Algorithmics (6EAP)

Graphs II – weighted graphs

Shortest paths

Jaak Vilo

2020 Fall
Shortest paths between nodes in graph

- Practical applications
- **Transportation**
  - Cheapest or quickest way to travel from A to B
- **Motion planning**
  - Most natural way for a cartoon character to navigate between places
- **Communications**
  - Time to send a message; diameter of a graph,...
Problem Definition

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

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

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

A shortest path may not exist.

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**
  - No algorithm for this problem are known that run asymptotically faster than the best single-source algorithm in the worst case
- **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

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.
Optimal Substructure of A Shortest-Path

- Lemma 24.1 (Subpath of shortest paths are shortest paths). Let \( p = \langle v_1, v_2, \ldots, v_k \rangle \) 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 \( p_{ij} = \langle v_{i1}, v_{i2}, \ldots, v_{ij} \rangle \) be the subpath of \( p \) from vertex \( v_i \) to \( v_j \). Then \( p_{ij} \) is a shortest path from vertex \( v_i \) to \( v_j \).

Examples of shortest paths depending on start node

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.

Bellman-Ford (1956-57-58)

Bellman-Ford (\( G, w, s \))
1. Initialise-Single-Source(\( G, s \))
2. for \( i = 1 \) to \( |G.V| - 1 \) /* \( n-1 \) rounds */
3. for each edge \( (u,v) \in G.E \)
4. RELAX( \( u, v, w \) )
5. for each edge \( (u,v) \in G.E \)
6. if \( v.d > u.d + w(u,v) \)
7. return FALSE
8. return TRUE
Bellman-Ford

- $O(VE)$
- Just repeatedly relax all edges.
  - Allow V cycles to propagate through the network
- Sedgwick – Algorithms


Queue-based **Bellman-Ford**

Specifically, we can easily determine a priori that numerous edges are not going to lead to a successful relaxation in any given pass: the only edges that could lead to a change in $\text{dist}\{v\}$ are those leaving a vertex whose $\text{dist}\{v\}$ value changed in the previous pass. To keep track of such vertices, we use a FIFO queue. The operation of the algorithm for our

**Algorithm 4.11 Bellman-Ford algorithm (queue-based)**

```java
public class BellmanFord {
    private double[] dist;
    private int[] edge[];
    private boolean[] marked;
    private int[] pred;
    public BellmanFord (int n) {
        dist = new double[n];
        edge = new int[n][n];
        marked = new boolean[n];
        pred = new int[n];
    }
    public void relax(int u, int v, int w) {
        if (dist[u] + w < dist[v]) {
            dist[v] = dist[u] + w;
            pred[v] = u;
            marked[u] = true;
        }
    }
    public void shortestPath() {
        for (int i = 0; i < dist.length; i++) {
            for (int j = 0; j < dist.length; j++) {
                relax(i, j, edge[i][j]);
            }
        }
    }
}
```

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)

Dijkstra’s Algorithm (1956, 1959)

- Solve the single-source shortest-paths problem on a weighted, directed graph when 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 ∈ V - S (kept in Q) with the minimum shortest-path estimate, add s u to S, and relax all edges leaving u

Dijkstra’s Algorithm (Cont.)

**DIJKSTRA(G, w, s)**

1. INITIALIZE-SINGLE-SOURCE(G, s)
2. S ← ∅
3. Q ← V[G]
4. while Q ≠ ∅
   5. do u ← EXTRACT-MIN(Q)
   6. S ← S ∪ {u}
   7. for each vertex v ∈ Adj[u]
      8. do RELAX(u, v, w)

Note: relax requires updating of min values in Q.
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 \( \mathcal{O}(V) \)
  - Line 7-8: for loop and relaxation \( \mathcal{O}(|E|) \)
  - Running time depends on how to implement min-priority queue
    - Simple array: \( \mathcal{O}(V^2 + E) \)
    - Binary min-heap: \( \mathcal{O}(V \log V) \)
    - Fibonacci min-heap: \( \mathcal{O}(V \log V + E) \)

http://www.cs.utexas.edu/users/EWD/

- **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

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
Dijkstra's Algo
1) Dijkstra is a Greedy based algorithm and similar to Prim's MST algo.
2) Dijkstra doesn't work for negative weight edges.
3) Time complexity of Dijkstra is $O(|E| + |V|\log|V|)$
4) Dijkstra's algorithm is usually the working principle behind link-state routing protocols, OSPF and IS-IS

Eucl- Path
We'll look more closely at this with the A* algorithm (heuristic search)
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

![Image of a graph with matrix operations]

Paths of length 2

![Image of paths of length 2]

\[ \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 2 & 0 & 1 & 0 & 0 & 0 \\ 3 & 0 & 0 & 1 & 0 & 1 \\ 4 & 0 & 0 & 0 & 0 & 1 \\ 5 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 2 & 0 & 1 & 0 & 0 & 0 \\ 3 & 0 & 0 & 1 & 0 & 1 \\ 4 & 0 & 0 & 0 & 0 & 1 \\ 5 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 2 & 0 & 1 & 0 & 0 & 0 \\ 3 & 0 & 0 & 1 & 0 & 1 \\ 4 & 0 & 0 & 0 & 0 & 1 \\ 5 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \]
\[ c_{ij} = \sum_k a_{ik} b_{kj} \]

\[ c_{st} = \sum_i a_{si} b_{it} \]

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] & A[i][t]) C[s][t]=true

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:

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 \( C = A \cdot B \). 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 \( a+b \) 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:

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;

Binary matrix “multiplication”

\( a+b \to a \text{ or } b \quad a*b \to a \text{ and } b \)

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 $G$.
Multiply:

```c
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] & A[i][t]  ) C[s][t]=true
```

Transitive closure:

```c
for( i=0 ; i<V ; i++ )
  for( s=0 ; s<V ; s++ )
    for( t=0 ; t<V ; t++ )
```

**Proof**

- **Proof:** transitive closure by induction on $i$.
  - **Iteration 1:** either $s$-$t$ or the path $s$-$0$-$t$.
  - **It 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 $i$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 $t$).

- 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 $i+1$ 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 $A[s][t]$ and $A[i+1][t]$ were previously set to true (by hypothesis), so the inner loop sets $A[s][t]$.

.paths via 0
paths via 1 (including 0-1, 1-0)
... paths via 1 (including 0,1,2,...)
Transitive closure:
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

• How to further improve this?

• Test for A[s][i] early

Program 19.3 Warshall’s algorithm

The constructor for class DenseGraph imposes the transitive closure of G in the private data field T so that classes can use Dijkstra’s objects to test whether any given vertex s in a digraph is reachable from any other given vertex. The constructor initializes T with a copy of G, adds self-loops, then runs Warshall’s algorithm to compute the transitive closure. We use a DenseGraph object for the transitive closure T because the algorithm needs an efficient implementation of the edge existence test (see Section 17.1).

class GraphIC
{
private DenseGraph T;
GraphIC(Graph G)
{
T = GraphUtilities.decompose(G);
for (int i = 0; i < T.V; i++)
T.insert(new Edge(a, a,));
for (int i = 0; i < T.V; i++)
for (int j = 0; j < T.V; j++)
T.insert(new Edge(i, j));
T.insert(new Edge(i, j));
}
checkReachable(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 the Warshall-like DFS, the running time grows up to a factor of 8 when we double V, which supports the conclusion that it is essentially proportional to V^3. The adjacency-list DFS takes time proportional to V^2 E, which explains the running time roughly as being a factor of 4 when we double both V and E (sparse graphs) and a factor of about 2 when we double E (dense graphs). All our algorithms were implemented in C and run on a Sun Sparcstation 20.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Sparse (V/1000 edges)</th>
<th>Dense (25 vertices)</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>W</td>
<td>L</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>128</td>
<td>19</td>
<td>23</td>
</tr>
<tr>
<td>256</td>
<td>275</td>
<td>46</td>
</tr>
<tr>
<td>512</td>
<td>550</td>
<td>120</td>
</tr>
<tr>
<td>1024</td>
<td>1050</td>
<td>250</td>
</tr>
</tbody>
</table>

Key:
W: Warshall’s algorithm (Section 19.3)
A: DFS, adjacency-list representation (Program 19.3)
B: DFS, adjacency-matrix representation (Programs 19.4 and 17.7)
C: DFS, adjacency-list representation (Program 19.4 and 17.7)

Graph:
• A>B>0.05
• A>C>0.05
• B>D>0.7
• B>E>0.3
• C>E>1.0
• D>A>1.0
• E>F>D>0.2
• E>G>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

Algorithmics/MarkovWalk
perl GraphMarkovChain.pl Weighted_Graph.txt
Matrix:
0 0.95 0.05 0 0
0 0 0 0.7 0.3
1.0 0 0 0 1.0
0.8 0 0 0 0

Random Walk with 10000 steps
FINAL: 0.000 0.000 0.000 0.000 1.000
Matrix multiplications with 10000 steps

Googe Page Rank

$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$

Equal probability 1/N (1-d) times
Sum of probabilities on all pages linking to pi (d times)

Historic snapshot

$\nu = \frac{0.85 \times 11}{11} + 0.15$
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

Module evaluation

- GO: Brain development
- Pigment granule
- Melanine metabolic process
- Transforming growth factor beta signaling pathway
- Embryonic development, gastrulation
- KEGG: Type II diabetes mellitus, cancers, WNT pathway

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

Stijn van Dongen PhD thesis
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
Fast Fully Dynamic Landmark-based Estimation of Shortest Path Distances in Very Large Graphs
Socially sensitive search

Naïve approach (Breadth-First-Search) requires 5-20 minutes

Landmark-based estimation

Basic Method

Mary Lee

1 <= d <= 7

3 <= d <= 5

Landmark-based estimation
Shortest path tree

Least common ancestor

Least common ancestor

Combining multiple landmarks

Combining multiple landmarks

Combining multiple landmarks

Landmarks-BFS
Combining multiple landmarks

**Landmarks-BFS**

Given two nodes $U$ and $V$:
1. Collect all paths from $U$ and $V$ to all landmarks
2. Run a BFS* on the induced subgraph

* or Dijkstra, or A*, or anything else

Landmark-based approximation

**Basic Method**

**LCA**

**Shortcutting**

**Landmarks-BFS**

Landmark-based approximation

**Dynamic**

**Basic Method**

**LCA**

**Shortcutting**

**Landmarks-BFS**

Insertion of an edge

**Dynamic**

**Basic Method**

**LCA**

**Shortcutting**

**Landmarks-BFS**
### Evaluation - Data

| Dataset | $|V|$ | $|E|$ | $d$ | $\Delta$ | $S/|V|$ | $t_{BFS}$ |
|---------|-----|-----|-----|--------|--------|--------|
| DBLP    | 770K| 2.6M| 6.3 | 23     | 85%    | 345 ms |
| Orkut   | 3.1M| 117M| 5.7 | 10     | 100%   | 8 sec  |
| Twitter | 41.7M| 1.2B| 4.2 | 24     | 100%   | 9 min  |
| Skype   | 454M| 3.1B| 6.5 | 59     | 85%    | 20 min |

### Timings - Query

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Method</th>
<th>No. of Landmarks</th>
<th>20</th>
<th>60</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skype</td>
<td>Basic</td>
<td>0.18</td>
<td>0.56</td>
<td>0.91</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LCA</td>
<td>1.06</td>
<td>2.43</td>
<td>3.69</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SC</td>
<td>1.22</td>
<td>2.92</td>
<td>4.85</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LBFS</td>
<td>5.10</td>
<td>13.24</td>
<td>16.25</td>
<td></td>
</tr>
</tbody>
</table>

Time for a batch of 500 queries / 500, in ms

- Linux, mmap, 32 cores, 256GB RAM
Timings: **Updates**

<table>
<thead>
<tr>
<th></th>
<th>DBLP</th>
<th>Orkut</th>
<th>Twitter</th>
<th>Skype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion</td>
<td>1μs</td>
<td>10μs</td>
<td>10μs</td>
<td>30μs</td>
</tr>
<tr>
<td>Deletion</td>
<td>100μs</td>
<td>2ms</td>
<td>12ms</td>
<td>11ms</td>
</tr>
</tbody>
</table>

* very non-uniform

Outline

- Improvement to Basic Landmark method
- Dynamic updates
  - Landmark selection
  - Evaluation

Landmark selection method

- Landmark is good if it covers many shortest paths

Best Coverage

- Highest degree
- Best coverage

Best Coverage

- A
- C
- E
- B
- D
- F

Best Coverage

- A
- C
- E
- B
- D
- F

Best Coverage

- A
- C
- E
- B
- D
- F
Results

- **Error**: \( d' - d \)

- **Timings**: Landmark selection

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Highest degree</th>
<th>Best coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLP</td>
<td>140 ms</td>
<td>2 min</td>
</tr>
<tr>
<td>Orkut</td>
<td>2 s</td>
<td>15 min</td>
</tr>
<tr>
<td>Twitter</td>
<td>22 s</td>
<td>15 hr</td>
</tr>
<tr>
<td>Skype</td>
<td>1 min</td>
<td>54 hr</td>
</tr>
</tbody>
</table>

- **Summary (Skype graph)**

<table>
<thead>
<tr>
<th></th>
<th>500M nodes</th>
<th>3B edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Landmark selection time (HD)</td>
<td>1 min / 54hr</td>
<td></td>
</tr>
<tr>
<td>Landmark computation time</td>
<td>20 min x 100</td>
<td></td>
</tr>
<tr>
<td>Total space for 100 landmarks</td>
<td>170G</td>
<td></td>
</tr>
<tr>
<td>Avg query time (SC/LBFS)</td>
<td>5ms / 16ms</td>
<td></td>
</tr>
<tr>
<td>Avg edge insertion time</td>
<td>0.030 ms</td>
<td></td>
</tr>
<tr>
<td>Avg edge deletion time</td>
<td>11 ms</td>
<td></td>
</tr>
<tr>
<td>Avg relative error (SC/LBFS)</td>
<td>18% / 15%</td>
<td></td>
</tr>
</tbody>
</table>

- **Questions**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LCA</td>
<td></td>
</tr>
<tr>
<td>Shortcutting</td>
<td></td>
</tr>
<tr>
<td>Landmarks-BFS</td>
<td></td>
</tr>
<tr>
<td>Dynamic updates</td>
<td></td>
</tr>
<tr>
<td>Highest degree</td>
<td></td>
</tr>
<tr>
<td>Best coverage</td>
<td></td>
</tr>
</tbody>
</table>

- **Timings: Query / Twitter**

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Method</th>
<th>No. of Landmarks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>20 60 100</td>
</tr>
<tr>
<td>Twitter</td>
<td>Basic</td>
<td>0.15 0.49 0.84</td>
</tr>
<tr>
<td></td>
<td>LCA</td>
<td>0.72 0.81 1.21</td>
</tr>
<tr>
<td></td>
<td>SC</td>
<td>0.89 0.99 1.87</td>
</tr>
<tr>
<td></td>
<td>LBFS</td>
<td>240 633 889</td>
</tr>
</tbody>
</table>
Generalizations

- To weighted graph:
  - Use weighted shortest path trees
  - The dynamic update algorithm becomes slightly more complicated

- To directed graph:
  - Use two SPTs per landmark

Improvements

- Parallelization possible at most stages
- "Evolutionary" on-line selection of landmarks
- Use of landmark-based heuristics with A* for exact path possible (Goldberg et al., Ikeda et al.)

Advanced Algorithmics (6EAP)
Graphs II – weighted graphs
Social networks

Jaak Vilo
2020 Fall

Community structure

- Citation networks
  - Group of similar research papers / researchers

- WWW
  - Set of closely related web pages

- Social networks
  - Facebook groups (group of people who are friends and go same school)

Community Structure of Zachary Karate club using Girvan Newman

FIG. 8: The network of friendships between individuals in the karate club study of Zachary [20]. The administrator and the instructor are represented by nodes 1 and 33, respectively. Shield squares represent individuals who ended up aligning with the club, while circles those who aligned with the instructor.
1. Divisive Methods

- Algorithm of Girvan-Newman
  - Calculate the betweenness of all edges
  - Cut the edge with highest betweenness
  - Recalculate edge betweenness
  - Repeat until no edges left or some criteria is met
- Complexity $O(m^2n)$ (m-edges, n-nodes)


Edge Betweenness

Edge Betweenness: the number of shortest paths between pairs of nodes that run along the edge.

Edge deletion? When do we stop

- Modularity – a measure of community goodness.
- Modularity $Q=\{(\text{edges inside the community})-(\text{expected number of edges inside the community for a random graph with same node degree distribution as the given network)}\}$

Modularity

- Modularity ranges from -1 to 1.
- It is positive if the number of edges inside the group are more than the expected number.
- Variation from 0 indicate difference with random case.
- Modularity can be used at each round of the Girvan-Newman algorithm to check if it is time to stop.

2. Modularity maximization: Louvain method

- First, it looks for ”small” communities by optimizing modularity in a local way.
- Second, it aggregates nodes of the same community and builds a new network whose nodes are the communities.
- These steps are repeated iteratively until a maximum of modularity is attained.

Belgian mobile phone network

- 2 million customers
- Red clusters are French speaking, Green clusters Dutch speaking

3. Link Clustering

- Community can be “a group of topologically similar links”
- Find communities by grouping edges that share a node into same community by hierarchical clustering

References

MAXIMUM FLOW

Mass-Flow Min-Cut Theorem (Ford Fulkerson's Algorithm)

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)

Capacity Flow

The Value of a Flow.

The value of a flow is given by:

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

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 Example](image)

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

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

- $x_{i,j}$ = flow on edge $(i,j)$
- $u_{i,j}$ = 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 $i \neq s,t$

$\sum_{j} x_{sj} = v$

$0 \leq x_{ij} \leq u_{ij}$ for all $(i,j) \in E.$

In simple terms maximize 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 $T$ ($T = V - S$) such that $s$ (source) is in $S$ and $t$ (target) is in $T.$

Capacity of Cut $(S,T)$

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

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 $|f| \leq \text{CAP}(S,T).$

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:

$$|f| = \sum_{e \in S \rightarrow T} f(e) - \sum_{e \in S \rightarrow T} f(e)$$

$$= \sum_{e \in S \rightarrow T} f(e) - \sum_{e \in S \rightarrow T} u(e)$$

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

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:
  \[ r(c(i,j)) = c(i,j) - f(i,j) \]
  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.

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 $s,v_1,e_1,v_2,e_2,...,v_k,t$ 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 the residual network are the edges on which the residual capacity is positive.
  - We call the maximum capacity by which we can increase the flow on $p$ the residual capacity of $p$.

- \[ c_f(p) = \min \{c_f(u,v) : (u,v) \text{ is on } p \} \]

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 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 $G$ 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 $nw G_r$ will have no more augmented paths.
  2. Let $v=F_x(S,T)$ 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.

Find any s-t path in G(x)

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

Find any s-t path

Determine the capacity $\Delta$ of the path.
Send $\Delta$ units of flow in the path.
Update residual capacities.
Find any s-t path

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

Find any s-t path

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

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.

These are the nodes that are reachable from node s.

Counterexample for termination

\[
\begin{align*}
\text{Clausene 54. Vaatluse voorg järgmised jooned.} \\
\text{Ri} & \rightarrow R^2 \\
\text{R} & \rightarrow \text{R}^2 \\
\end{align*}
\]

\[
\text{Examples: } R = \frac{1}{2} (R^2 + R^{-1}) \text{ ja fikseeritud struktuur}
\]

- \[
1. s \rightarrow c \rightarrow f \rightarrow d \rightarrow b \rightarrow e \rightarrow t
\]
- \[
2. s \rightarrow b \rightarrow e \rightarrow f \rightarrow c \rightarrow a \rightarrow d \rightarrow t
\]
- \[
3. s \rightarrow a \rightarrow d \rightarrow c \rightarrow b \rightarrow c \rightarrow f \rightarrow t.
\]

Näita, et Ford-Fulkersoni algoritm ei lõpeta tööd.

Distribution & Transportation

Assigning teachers to classes

Teacher likes to teach C1, C4, C6
Every course will need a nr of teachers
Every teacher has a maximal capacity to teach
"Likes" – by weight

How would you solve it?
Job placement: 6 people, 6 jobs, preferences...

Alice  Adobe  Alice  Adobe
Apple  Bob  Dave  Apple
HP  Carol  HF  Frank
Bob  IBM  Carol  Frank
Yahoo  Enka  Sun  Yahoo
Carol  IBM  Carol  Frank
Yahoo  Bob  IBM  Frank

Converting the Matching problem to Network Flow

Converting Optimal Bipartite Matching to Network Flow