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

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

• 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.

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**

Images: [http://www.personal.kent.edu/~rmuhamma](http://www.personal.kent.edu/~rmuhamma)
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 $A$

**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

**Illustration of Theorem 23.1**

- $A = \{(a, b), (c, i), (h, d), (g, f)\}$
- $S = \{a, b, c, d, g, f\}$
- Many kinds of cuts satisfying the requirements of Theorem 23.1
- $(c, f)$ is the light edges crossing $S$ and $V - S$

**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 $(x, y) \notin A \Rightarrow A \subseteq T'$
  - $A \cup \{(u, v)\} \subseteq T'$

**GENERIC-MST**

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

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

Properties of GENERIC-MST

- As the algorithm proceeds, the set $A$ is always acyclic
- $G_n(V, A)$ is a forest, and each of the connected component of $G_n$ is a tree
- Any safe edge $(u, v)$ for $A$ connects distinct component of $G_n$, 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 $G_n = (V, A)$. If $(u, v)$ is a light edge connecting $C$ to some other components in $G_n$, then $(u, v)$ is safe for $A$
**Prim’s Algorithm in Action**

![Graph examples]

**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
- \( A = \{(v, \pi[v]) : v \in V - \{r\} - Q\} \) \( \Rightarrow \) finally \( Q = \emptyset \)

---

**Prim’s Algorithm**

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

**MST-\text{PRIM}(G, w, r)**

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

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)
- Use Fibonacci heap to implement the min-priority queue Q
  - O(E + V \lg V)

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)
- 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.

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 leaves 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 \(O(m \alpha(m))\), 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 to 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 \(\text{union}\) and \(\text{find}\) 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 \(\text{find}(j)\).

This path compression will let us do better than \(O(n \log n)\) for \(n\) union-finds.
\(O(\alpha(n))\) Not quite. Difficult analysis shows that it takes \(O(\alpha(n))\) 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.

**Table 24.1 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 $K$ is no smaller than $V - 1$) and that there are $X$ edges not longer than the longest edge in the MST (see Property 24.10). These worst-case bounds may be too conservative to be useful in predicting performance on real graphs. The algorithms run in real-time as 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 (PQ, heap)</td>
<td>$E \log V$</td>
<td>conservative upper bound</td>
</tr>
<tr>
<td>Prim (PQ, u-heap)</td>
<td>$E \log V$</td>
<td>linear unless extremely sparse</td>
</tr>
<tr>
<td>Kruskal (partial)</td>
<td>$E \cdot \log V$</td>
<td>sort cost dominates</td>
</tr>
<tr>
<td>Boruvka</td>
<td>$E \cdot V$</td>
<td>conservative upper bound</td>
</tr>
</tbody>
</table>

**SINGLE-SOURCE SHORTEST PATHS**

(CHAPTER 24)

4-letter words, distance 1
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...

Example: Predictive Mobile text Entry Messaging...

What was the message?

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. ‘hiver 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:
  
  $$w(p) = \sum_{i=1}^{k} w(v_i, v_{i+1})$$

- 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 paths 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

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 $p_{ij} = <v_1, v_2, ..., v_i, v_j>$ 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
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** $(G, s)$

- $d[v] \leftarrow \infty$ for each vertex $v \in V(G)$
- $\pi[v] \leftarrow \text{NIL}$
- $d[s] \leftarrow 0$

**Relaxation**

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 $\pi[v]$.

**Bellman-Ford**

- $O(V E)$
- Just repeatedly relax all edges.
  - Allow $V$ cycles to propagate through the network

**Shortest paths on a DAG**

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

$O(V + E)$
Dijkstra’s Algorithm

• 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 to S, and relax all edges leaving u

Dijkstra’s Algorithm (Cont.)

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

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

Note: relax requires updating of min values in Q.

Figure 214 The execution of Dijkstra’s algorithm. The source s is the bottom vertex. The shortest path estimates are shown in the vertices, and shaded edges indicate predecessor values. Black vertices are in the set S, and white vertices are in the incompletely solved \( Q = V - S \).

(a) The situation just before the first iteration of the while loop of lines 4-6. The shaded vertex has the estimate s in line 3. (b) The situation just before the second iteration of the while loop of lines 4-6. The shaded vertex has the estimate s in line 3. (c) The situation just after the third iteration of the while loop of lines 4-6. The shaded vertex has the estimate s in line 3. (d) The situation just after the fourth iteration of the while loop of lines 4-6. The shaded vertex has the estimate s in line 3. (e) The situation just after the fifth iteration of the while loop of lines 4-6. The shaded vertex has the estimate s in line 3. (f) The situation just after the sixth iteration of the while loop of lines 4-6. The shaded vertex has the estimate s in line 3. (g) The situation just after the seventh iteration of the while loop of lines 4-6. The shaded vertex has the estimate s in line 3. (h) The situation just after the eighth iteration of the while loop of lines 4-6. The shaded vertex has the estimate s in line 3. (i) The situation just after the ninth iteration of the while loop of lines 4-6. The shaded vertex has the estimate s in line 3. (j) The situation just after the tenth iteration of the while loop of lines 4-6. The shaded vertex has the estimate s in line 3. (k) The situation just after the eleventh iteration of the while loop of lines 4-6. The shaded vertex has the estimate s in line 3. (l) The situation just after the twelfth iteration of the while loop of lines 4-6. The shaded vertex has the estimate s in line 3. (m) The situation just after the thirteenth iteration of the while loop of lines 4-6. The shaded vertex has the estimate s in line 3.
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: $O((V+E)\log V)$
    - Fibonacci min-heap: $O(V\log V + E)$

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

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

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

ABSTRACT

Computing the shortest path between a pair of vertices in a graph is a fundamental problem in graph algorithms, with applications in various fields. Existing solutions are not sufficient in contemporary, rapidly evolving social networks with huge-scale graphs.

Keywords

Graph Databases, Shortest Paths, Social Networks, Landmark-based, Time, Dynamic Updates

ACM Conference on Information and Knowledge Management (CIKM) 2011
Naïve approach (Breadth-First-Search) requires 5-20 minutes.

Landmark-based estimation:

Basic Method

Mary Lee

Mary Ann

3 <= d <= 5
6.4.2012

Landmark-based estimation

Least common ancestor

Combining multiple landmarks

Shortest path tree

Least common ancestor

Combining multiple landmarks
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**

Accuracy vs Speed

Dynamic

Insertion of an edge
Results

Deletion – more complicated

Evaluation - Data

| Dataset  | |V| |E| |d| |Δ| |S/V| |tap₂| |
|----------|---|---|---|---|---|---|---|---|---|---|
| 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 |

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Graph file</th>
<th>Landmark file</th>
<th>Basic</th>
<th>LCA/SC/LBFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLP</td>
<td>27M</td>
<td>75M</td>
<td>3.0M</td>
<td></td>
</tr>
<tr>
<td>Orkut</td>
<td>935M</td>
<td>493M</td>
<td>3.0M</td>
<td></td>
</tr>
<tr>
<td>Twitter</td>
<td>3.2G</td>
<td>17.0M</td>
<td>1.7G</td>
<td></td>
</tr>
<tr>
<td>Skype</td>
<td>27G</td>
<td>433M</td>
<td>1.7G</td>
<td></td>
</tr>
</tbody>
</table>

*per each landmark*
Timings: Query

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Method</th>
<th>No. of Landmarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skype</td>
<td>Basic</td>
<td>0.18, 0.56, 0.91</td>
</tr>
<tr>
<td></td>
<td>LCA</td>
<td>1.06, 2.43, 3.69</td>
</tr>
<tr>
<td></td>
<td>SC</td>
<td>1.22, 2.92, 4.85</td>
</tr>
<tr>
<td></td>
<td>LBFS</td>
<td>5.10, 13.24, 16.25</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>Dataset</th>
<th>DBLP</th>
<th>Orkut</th>
<th>Twitter</th>
<th>Skype</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Insertion</td>
<td>1μs</td>
<td>10μs</td>
<td>10μs</td>
</tr>
<tr>
<td></td>
<td>Deletion*</td>
<td>100μs</td>
<td>2ns</td>
<td>12ns</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
- Highest degree
- Best coverage

Best Coverage

```
A
B
C
D
E
F
```
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 h</td>
</tr>
<tr>
<td>Skype</td>
<td>1 min</td>
<td>54 h</td>
</tr>
</tbody>
</table>

Summary (Skype graph)

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

Questions

- LCA
- Shortcutting
- Landmarks-BFS
- Dynamic updates
  - Highest degree
  - Best coverage
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.)

Timings: Query / Twitter

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Method</th>
<th>No. of Landmarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Twitter</td>
<td>Basic</td>
<td>20 0.15 0.49 0.84</td>
</tr>
<tr>
<td></td>
<td>LCA</td>
<td>60 0.72 0.81 1.21</td>
</tr>
<tr>
<td></td>
<td>SC</td>
<td>100 0.82 0.99 1.87</td>
</tr>
<tr>
<td></td>
<td>LBFS</td>
<td></td>
</tr>
</tbody>
</table>

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

\[ c_{ij} = \sum_k a_{ik} b_{kj} \]

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

---

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 \( i = 0; C[s][i] = 0; i < V; i++ \)
      \[ C[s][t] += A[s][i] \cdot B[i][t]; \]
Diagonal 1 = self-loop

Diagonal 0 or 1

G*G

Transitive closure

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

Book: Sedgewick, Algorithms ...

- 19.3. Reachability and Transitive Closure
  - [link 1](#)
  - [link 2](#)
  - [link 3](#)
Complexity...

- for \( i = 1 \) to \(|V|\) do \( V^0 = V^{i-1} \cdot V\)
- \( V^3 \) operations for \( V^2, V^3, \ldots V^v \)
- \( \Rightarrow O(V^4) \)
- Use exponential: 2 \( \Rightarrow 4 \) \( \Rightarrow 8 \) \( \Rightarrow 16 \) \( \Rightarrow \) steps.
- \( V^2 \cdot V^2 = V^4, V^4 \cdot V^2 = V^6, \ldots \Rightarrow O(\log V) \cdot V^3 \)
- Can we avoid so many cycles?

Multiply:

```
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];
```

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;
```

Proof:

- Proof: transitive closure by induction on \( i \).
  - Iteration 1: either \( s \)-\( t \) or the path \( s-0-1-t \).
  - It 2: all the paths between \( s \) and \( t \) that include 1 and perhaps 0, such as \( s-1-0-1-t \) and \( s-0-1-0-1-t \).
  - 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 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].

Program 19.3. Warshall’s algorithm

The constructor for class GraphT computes the transitive closure of G in the private data field T so the classes can use GraphT objects to see whether are given vertices a is reachable from any other given vertex. The constructor initializes T with a copy of G, while method computeT() makes its own copy of T and runs Warshall’s algorithm on the copy. We use a Deque-based algorithm for the transitive closure T because the algorithm and its efficient implementation of the edge counter are less sensitive to T.

class GraphT
private DGraph T;
GraphT(Graph G) {
  T = GraphUtilities.deepcopy(G);
  T.insert(new Edge(a, a));
  for (int i = 0; i < T.V(); ++i) {
    for (int j = 0; j < T.E(); ++j)
      T.insert(new Edge(a, i));
  }
  boolean reachable(int s, int t) {
    return T.contains(s, t);
  }
}

Random walks...

Matrix:

Table 19.1: Empirical study of transitive-closure algorithms

How to further improve?

Test for A[s][i] early
Finding the modules

Matrix:

0 0.85 0.05 0 0
0 0 0.7 0.3 0
0 0 0 1 0
0 0 0 0 1
0 0 0 0 0

Random Walk with 10000 steps

Frank: 0.322596 0.230774 0.016714 0.22768 0.10214

Finding the modules

Matrix:

0.320561516139543 0.320561516139543 0.320561516139543 0.320561516139543 0.320561516139543
0.320561516139543 0.320561516139543 0.320561516139543 0.320561516139543 0.320561516139543
0.320561516139543 0.320561516139543 0.320561516139543 0.320561516139543 0.320561516139543
0.320561516139543 0.320561516139543 0.320561516139543 0.320561516139543 0.320561516139543
0.320561516139543 0.320561516139543 0.320561516139543 0.320561516139543 0.320561516139543

Finding the modules

Module evaluation

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

MCL clustering algorithm

• Markov (Chain Monte Carlo) Clustering

• Random walks according to edge weights

• Follow the different paths according to their probability

• Regions that are traversed “often” form clusters
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 \( E \).

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

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

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:

| 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

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

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)

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

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:

\[ |f| = \sum_{e \in S} f(e) - \sum_{e \in T} f(e) \]

\[ = \sum_{e \in S} u(e) - \sum_{e \in T} u(e) \]

\[ = \sum_{e \in S} u(e) - \sum_{e \in T} u(e) \]

\[ = \text{CAP}(S,T) \]
Methods

Max-Flow Min-Cut Theorem

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

The Ford-Fulkerson Method

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}.

The Ford-Fulkerson's Algorithm

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 i iterations.
Consider augmentation i along path P. We update the residual capacities by 0 or Δ. 
The residual capacity Δ of P is the smallest residual capacity on P.
After updating, we modify the residual capacities by 0 or Δ.
and thus residual capacities stay integers.

Theorem: Ford-Fulkerson’s algorithm is finite
Proof: The capacites of each augmenting path is at most 1. The augmentation reduces the residual capacity of some edge (s,i) by Δ and doesn’t increase the residual capacity for some edge (i,j).
So the sum of residual capacities of edges out of s keeps decreasing, and is bounded below 0.
Number of augmentations is O(kC) where C is the largest of the capacity in the network.

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, v1, w1, v2, w2, ..., vt, 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.

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.

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 i augmentations, and consider augmentation i along path P.
The residual capacity Δ of P is the smallest residual capacity on P.
After updating, we modify the residual capacities by 0 or Δ, 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 decreases the residual capacity of some edge (s,i) by Δ and doesn’t increase the residual capacity for some edge (i,j) for any j.
So the sum of residual capacities of edges out of s keeps decreasing, and is bounded below 0.
Number of augmentations is O(kC) 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 $v = F_x(S, T)$ 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)
Ford-Fulkerson Max Flow

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.

Counterexample for termination

Distribution & Transportation
Job placement:
6 people, 6 jobs, preferences...

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?