Weighted Graph Algorithms

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

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

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

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.

![Diagram](image1)

Note: There can be more than one minimum spanning tree coincident as a group with identical weight edges.

Fully connected graph.
Find a MST?
MST

Images: http://www.personal.kent.edu/~rmuhamma/

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**

$\text{GENERIC-MST}(G, w)$

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 $A$

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

- **Theorem.** Let $A$ be a subset of $E$ that is included in some MST, let $(S, V-S)$ be any cut of $G$ that respects $A$, and let $(u, v)$ be a light edge crossing $(S, V-S)$. Then edge $(u, v)$ is safe for $A$
  
  - **Cut** $(S, V-S)$: a partition of $V$
  
  - **Crossing edge**: one endpoint in $S$ and the other in $V-S$
  
  - A cut respects a set of $A$ of edges if no edges in $A$ crosses the cut
  
  - A light edge crossing a cut if its weight is the minimum of any edge crossing the cut

**Figure 23.2** Two ways of viewing a cut $(S, V-S)$ of the graph from Figure 23.1. (a) The vertices in the set $S$ are shown in black, and those in $V-S$ are shown in white. The edges crossing the cut are those connecting white vertices with black vertices. The edge $(d,c)$ is the unique light edge crossing the cut. A subset $A$ of the edges is shaded; note that the cut $(S, V-S)$ respects $A$, since no edge of $A$ crosses the cut. (b) The same graph with the vertices in the set $S$ on the left and the vertices in the set $V-S$ on the right. An edge crosses the cut if it connects a vertex on the left with a vertex on the right.
Illustration of Theorem 23.1

- \( A = \{(a, b), (c, i), (h, g), (g, f)\} \)

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

- \((c, f)\) is the light edges crossing \( S \) and \( V \setminus S \) 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
  - \( T' = T \setminus \{(x, y)\} \cup (u, v) \)
  - \( T' \) is also a MST since \( W(T') = W(T) - w(x, y) + w(u, v) \leq W(T) \)

- \((u, v)\) is actually a safe edge for \( A \)
  - Since \( A \subseteq T \) and \((x, y) \notin A \rightarrow A \subseteq T' \)
  - \( \Rightarrow A \cup \{(u, v)\} \subseteq T' \)
Properties of GENERIC-MST

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

---

**Figure 23.3**  The proof of Theorem 23.1. The vertices in $S$ are black, and the vertices in $V - S$ are white. The edges in the minimum spanning tree $T$ are shown, but the edges in the graph $G$ are not. The edges in $A$ are shaded, and $(u, v)$ is a light edge crossing the cut $(S, V - S)$. The edge $(x, y)$ is an edge on the unique path $p$ from $u$ to $v$ in $T$. A minimum spanning tree $T'$ that contains $(u, v)$ is formed by removing the edge $(x, y)$ from $T$ and adding the edge $(u, v)$.
The Algorithms of Kruskal and Prim

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

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

Prim’s Algorithm

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

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

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

As a greedy algorithm, which edge should we pick? The cheapest edge with which can grow the tree by one vertex without creating a cycle.

**Prim’s Algorithm (Pseudocode)**

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

Prim-MST(G)

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 $T_{prim}$.

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 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 = empty
Prim’s Algorithm

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

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

MST-PRIM($G, w, r$)

1. for each $u \in V[G]$
do
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. $\text{while } Q \neq \emptyset$
do
7.     $u \leftarrow \text{EXTRACT-MIN} (Q)$
8.     for each $v \in \text{Adj}[u]$
do
9.        if $v \in Q \text{ and } w(u,v) < \text{key}[v]$
10.           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, \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 NIL, then 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-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
    • EXTRACT-MIN: O(lg V)
  – The for loop in lines 8-11 is executed O(E) times altogether
    • Line 11: DECREASE-KEY operation: O(lg V)
  – Total performance = O(V lg V + E lg V) = O(E lg V)

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

Why is Prim Correct?
We use a proof by contradiction:
Suppose Prim’s algorithm does not always give the minimum cost spanning tree on some graph.
If so, there is a graph on which it fails.
And if so, there must be a first edge \((x, y)\) 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.

**Fast Kruskal Implementation**

Put the edges in a heap

```plaintext
count = 0
while (count < n − 1) do
    get next edge (v, w)
    if (component (v) ≠ component(w))
        add to T
        component (v)=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 {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 to 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.
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 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...

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

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

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

Prim vs Kruskal vs Boruvka
### Table 20.1 Cost of MST algorithms

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

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

---

**Advanced Algorithmics (6EAP)**

**Graphs II – weighted graphs**

**Shortest paths**

Jaak Vilo

2020 Fall
SINGLE-SOURCE SHORTEST PATHS (CHAPTER 24)

4-letter words, distance 1

LEDA tutorial
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?
Problem Definition

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

- We define the **shortest-path weight from $u$ to $v$** by
  \[ \delta(u, v) = \min \{ w(p) : u \xrightarrow{p} v \} \]  
  If there is a path from $u$ to $v$,
  Otherwise.

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

- **Single-source shortest paths** problem — greedy
  - Finds all the shortest path of vertices reachable from a single source vertex \( s \)

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

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

- **All-pairs shortest-path** problem — dynamic programming
  - Can be solved faster than running the single-source shortest-path problem for each vertex
Optimal Substructure of A Shortest-Path

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

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

![Graph Diagram](image)

Relaxation

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

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

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 \(\pi[v]\)

By Triangle Inequality.

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
- Sedgewick – Algorithms

Queue-based Bellman-Ford

Specifically, we can easily determine \textit{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 distTo[] are those leaving a vertex whose distTo[] 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 BellmanFordDP
{
    private double[] distTo; // length of path to v
    private DirectedEdge[] edgeTo; // last edge on path to v
    private boolean[] memo; // is this vertex on the queue?
    private Queue<Integer> queue; // vertices being relaxed
    private int cycle; // number of calls to relax()

    private Iterable<DirectedEdge> cycle; // negative cycle in edgeTo[]?

    public BellmanFordDP(Digraph G, int s)
    {
        distTo = new double[G.V()];
        edgeTo = new DirectedEdge[G.V()];
        memo = new boolean[G.V()];
        queue = new Queue<Integer>();
        distTo[s] = Double.POSITIVE_INFINITY;
        distTo[s] = 0.0;
        queue.enqueue(s);
        while (queue.size() > 0 && hasNegativeCycle())
        {
            int v = queue.dequeue();
            memo[v] = false;
            relax(G, v);
        }
    }

    private void relax(Digraph G, int v)
    // See page 673.
    public double distTo(int v) // standard client query methods
    public boolean hasPathTo(int v) // for SPT implementations
    public Iterable<DirectedEdge> pathTo(int v) // (See page 649.)
    public void findNegativeCycle()
    public boolean hasNegativeCycle()
    public Iterable<DirectedEdge> negativeCycle()
    // See page 677;
}
```
from currency $v$ to currency $w$, and any negative cycle is an arbitrage opportunity.

**Arbitrage in currency exchange**

**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 $v \in G.Adj[u]$**
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 $u \in V - S$ (kept in Q) with the minimum shortest-path estimate, add $s u$ to $S$, and relax all edges leaving $u$
Dijkstra's Algorithm (Cont.)

\[
\text{Dijkstra}(G, w, s) \\
1 \quad \text{Initialize-Single-Source}(G, s) \\
2 \quad S \leftarrow \emptyset \\
3 \quad Q \leftarrow V[G] \\
4 \quad \textbf{while } Q \neq \emptyset \\
5 \quad \quad \textbf{do } u \leftarrow \text{Extract-Min}(Q) \\
6 \quad \quad \quad S \leftarrow S \cup \{u\} \\
7 \quad \quad \textbf{for each vertex } v \in \text{Adj}[u] \\
8 \quad \quad \quad \textbf{do } \text{Relax}(u, v, w)
\]

Note: relax requires updating of min values in Q.
Figure 24.6 The execution of Dijkstra's algorithm. The source $s$ is the leftmost vertex. The shortest-path estimates are shown within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set $S$, and white vertices are in the min-priority queue $Q = V - S$.

(a) The situation just before the first iteration of the while loop of lines 4–8. The shaded vertex has the minimum $d$ value and is chosen as vertex $u$ in line 5. (b)–(f) The situation after each successive iteration of the while loop. The shaded vertex in each part is chosen as vertex $u$ in line 5 of the next iteration. The $d$ and $\pi$ values shown in part (f) are the final values.
Analysis of Dijkstra’s Algorithm

• Correctness: Theorem 24.6 (Loop invariant)
• Min-priority queue operations
  – INSERT (line 3)
  – EXTRACT-MIN (line 5)
  – DECREASE-KEY (line 8)
• Time analysis
  – Line 4-8: while loop $\Rightarrow O(V)$
  – Line 7-8: for loop and relaxation $\Rightarrow |E|$
  – Running time depends on how to implement min-priority queue
    • Simple array: $O(V^2 + E) = O(V^2)$
    • Binary min-heap: $O((V+E)\lg V)$
    • Fibonacci min-heap: $O(V\lg V + E)$

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
Eucl- Path

We’ll look more closely at this with the A* algorithm (heuristic search)

Explored area

Nodes that are not fully processed are in the “queue”

- queue (FIFO)
- stack (LIFO)
- priority queue
- random queue
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

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
0 & 0 & 0 & 1 & 0 & 0 \times 1 \\
1 & 1 & 0 & 0 & 0 & 0 \times 0 \\
2 & 0 & 1 & 0 & 0 & 0 \times 0 \\
3 & 0 & 0 & 1 & 0 & 1 \times 0 \\
4 & 0 & 0 & 0 & 0 & 1 \times 1 \\
5 & 0 & 0 & 0 & 0 & 1 \times 0 \\
\end{array}
\]
Paths of length 2

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
0 & 1 & 2 & 3 & 4 & 5 \\
\hline
0 & 0 & 0 & 1 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 \\
2 & 0 & 1 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 1 & 0 \\
4 & 0 & 0 & 0 & 0 & 1 \\
5 & 0 & 0 & 0 & 0 & 1 \\
\hline
\end{array}
\times
\begin{array}{|c|c|c|c|c|c|}
\hline
0 & 1 & 2 & 3 & 4 & 5 \\
\hline
0 & 0 & 0 & 1 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 \\
2 & 0 & 1 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 1 & 0 \\
4 & 0 & 0 & 0 & 0 & 1 \\
5 & 0 & 0 & 0 & 0 & 1 \\
\hline
\end{array}
= \\
\begin{array}{|c|c|c|c|c|c|}
\hline
0 & 1 & 2 & 3 & 4 & 5 \\
\hline
0 & 0 & 1 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 & 1 \\
2 & 1 & 0 & 0 & 0 & 0 \\
3 & 0 & 1 & 0 & 0 & 1 \\
4 & 0 & 0 & 0 & 1 & 0 \\
5 & 0 & 0 & 0 & 0 & 1 \\
\hline
\end{array}
\[ c_{ij} = \sum_{k} a_{ik} b_{kj} \]
\[ c_{st} = \sum_{i} a_{si} b_{it} \]

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

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 \( C = A \times B \). This operation is defined for matrices comprising any type of entry for which 0, +, and * are defined. In particular, if the matrix entries are either \text{true} or \text{false} and we interpret \text{a+b} to be the logical or operation and \text{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”
\[ a + b \to a \textbf{ or } b \quad a \times b \to a \textbf{ and } b \]

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

Diagonal 1 = self-loop

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

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

\[ G \ast G \ast G \ast \ldots \]

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

Exists link via \( j \)

1 on diagonal - link to itself

**Figure 19.13 Transitive closure**

This digraph (top) has just eight directed edges, but its transitive closure (bottom) shows that there are directed paths connecting 19 of the 30 pairs of vertices. Structural properties of the digraph are reflected in the transitive closure. For example, rows 0, 1, and 2 in the adjacency matrix for the transitive closure are identical (as are columns 0, 1, and 2) because those vertices are on a directed cycle in the digraph.

**Figure 19.14 Squaring an adjacency matrix**

If we put 1's on the diagonal of a digraph's adjacency matrix, the square of the matrix represents a graph with an edge corresponding to each path of length 2 (top). If we put 1's on the diagonal, the square of the matrix represents a graph with an edge corresponding to each path of length 1 or 2 (bottom).

**Property 19.6** We can compute the transitive closure of a digraph by constructing the latter's adjacency matrix, \( A \), adding self-loops for every vertex, and computing \( A^3 \).

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

• **19.3. Reachability and Transitive Closure**

Complexity...

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

• Use exponential: 2 \( \Rightarrow 4 \Rightarrow 8 \Rightarrow 16 \ldots \) steps.
• \( V^2 \times V^2 = V^4, V^4 \times V^4 = V^8, \ldots \Rightarrow O\left(\left\lfloor \log V \right\rfloor \times V^3 \right) \)
• Can we avoid so many cycles?
Multiply:

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

Transitive closure:

\[
\begin{align*}
&\text{for (} i = 0 ; i < V ; i++ \) \\
&\quad \text{for (} s = 0 ; s < V ; s++ \) \\
&\quad \quad \text{for (} t = 0 ; t < V ; t++ \) \\
&\quad \quad \quad \text{if (} A[s][i] \& A[i][t] \) \quad A[s][t]=true
\end{align*}
\]
Transitive closure:
\[
\text{for( } i=0 ; \ i<V ; \ i++ \ )
\text{ for( } s=0 ; \ s<V ; \ s++ \ )
\text{ for( } t=0 ; \ t<V ; \ t++ \ )
\text{ if( } A[s][i] \ & \ A[i][t] \ ) \ A[s][t]=true
\]

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

• Proof: transitive closure by induction on i.
• Iteration 1: either s-t or the path s-0-t.
• Iteration 2: all the paths between s and t that include 1 and perhaps 0, such as s-1-t, s-1-0-t, and s-0-1-t.
• Inductive hypothesis: The ith iteration of the loop sets the bit (s, t) to true iff there is a directed path from s to t in the digraph that does not include any vertices with indices greater than i (except possibly the endpoints s and t).

• Assuming that it is true for the ith iteration of the loop, there is a path from s to t that does not include any vertices with indices greater than i+1 iff
  – (i) there is a path from s to t without indices >i, in which case A[s][t] was set on a previous iteration of the loop (inductive hypothesis)
  – (ii) there is a path from s to i+1 and a path from i+1 to t, neither of which includes any vertices with indices greater than i (except endpoints), in which case A[s][i+1] and A[i+1][t] were previously set to true (by hypothesis), so the inner loop sets A[s][t]. ▪
• How to further improve this?

• Test for $A[s][i]$ early

Transitive closure:

\[
\begin{align*}
&\text{for } ( i=0 \ ; \ i<V \ ; \ i++) \\
&\quad \text{for } ( s=0 \ ; \ s<V \ ; \ s++) \\
&\quad \quad \text{next } s \text{ unless } A[s][i] \\
&\quad \text{for } ( t=0 \ ; \ t<V \ ; \ t++) \\
&\quad \quad \text{if } ( A[i][t] ) \ A[s][t]=\text{true}
\end{align*}
\]
Program 19.3  Warshall's algorithm
The constructor for class GraphTC computes the transitive closure of a digraph G in the private data field T so that clients can use GraphTC objects to test whether any given vertex in a digraph is reachable from any other given vertex. The constructor initializes T with a copy of G, adds self-loops, and then uses 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 17.5).

```java
class GraphTC
{
    private DenseGraph T;
    GraphTC(Graph G)
    {
        T = GraphUtilities.densecopy(G);
        for (int s = 0; s < T.V(); s++)
            T.insert(new Edge(s, s));
        for (int i = 0; i < T.V(); i++)
            for (int s = 0; s < T.V(); s++)
                if (T.edge(s, i))
                    for (int t = 0; t < T.V(); t++)
                        if (T.edge(i, t))
                        T.insert(new Edge(s, t));
        boolean reachable(int s, int t)
        { return T.edge(s, t); }
    }
}
```

Table 19.1  Empirical study of transitive-closure algorithms
This table shows running times that exhibit dramatic performance differences for various algorithms for computing the transitive closure of random digraphs, both dense and sparse. For all but the adjacency-lists DFS, the running time goes up by a factor of 8 when we double V, which supports the conclusion that it is essentially proportional to V^3. The adjacency-lists DFS takes time proportional to V|E|, which explains the running time roughly increasing by a factor of 4 when we double both V and E (sparse graphs) and by a factor of about 2 when we double E (dense graphs), except that list-traversal overhead degrades performance for high-density graphs.

<table>
<thead>
<tr>
<th>sparse (100 edges)</th>
<th>dense (250 vertices)</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>W</td>
</tr>
<tr>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>50</td>
<td>3</td>
</tr>
<tr>
<td>125</td>
<td>35</td>
</tr>
<tr>
<td>250</td>
<td>275</td>
</tr>
<tr>
<td>500</td>
<td>2222</td>
</tr>
</tbody>
</table>

Key:
- W  Warshall's algorithm (Section 19.3)
- W* Improved Warshall's algorithm (Program 19.3)
- A  DFS, adjacency-matrix representation (Programs 19.4 and 17.7)
- L  DFS, adjacency-lists representation (Program 19.4 and 17.9)
Diagonal 0 or 1  \quad G^*G

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

vilo@muhu:~/Algorithmics/MarkovWalk$ perl GraphMarkovChain.pl Weighted_Graph.txt
Matrix:
  0 0.95 0.05 0 0
  0 0 0 0.7 0.3
  0 0 0 0 1.0
  1.0 0 0 0 0
  0.8 0 0 0.2 0
Random Walk with 10000 steps
FINAL: 0.326051516139543 0.309748940332566 0.0163025758069772
        0.238669709814146 0.109227257906747
Matrix multiplications with 10000 steps
Matrix:
  0.326051516139543 0.309748940332566 0.0163025758069772 0.238669709814146 0.109227257906747
  0.326051516139543 0.309748940332566 0.0163025758069772 0.238669709814146 0.109227257906747
  0.326051516139543 0.309748940332566 0.0163025758069772 0.238669709814146 0.109227257906747
  0.326051516139543 0.309748940332566 0.0163025758069772 0.238669709814146 0.109227257906747
  0.326051516139543 0.309748940332566 0.0163025758069772 0.238669709814146 0.109227257906747
vilo@muhu:~/Algorithmics/MarkovWalk$
Google 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 $1/N$ \hspace{1em} (1-d) times
Sum of probabilities on all pages linking to $p_i$ \hspace{1em} (d times)
Historic snapshot

http://www.nowherenearithaca.com/2013/04/explorating-googles-pagerank.html
Finding the modules

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

Public datasets for *H. sapiens*
- 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: Transforming growth factor beta signaling pw.
- embryonic development, gastrulation
- KEGG: Cell cycle, cancers, WNT pw.

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

GO: Brain development
- Pigment granule
- Melanine metabolic process
MCL clustering algorithm

Stijn van Dongen

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

- Random walks according to edge weights

- Follow the different paths according to their probability

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

With this, the MCL algorithm can be written as

\[
\text{G is a graph} \\
\text{add loops to G} \quad \# \text{ see below} \\
\text{set } \Gamma \text{ to some value} \quad \# \text{ affects granularity} \\
\text{set } M_1 \text{ to be the matrix of random walks on } G \\
\text{while (change) } \\
\quad M_2 = M_1 + M_1 \quad \# \text{ expansion} \\
\quad M_1 = \Gamma(M_2) \quad \# \text{ inflation} \\
\quad \text{change} = \text{difference}(M_1, M_2) \\
\] \\
\text{set CLUSTERING as the components of } M_1 \quad \# \text{ see below}
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

Advanced Algorithmics (6EAP)
Graphs II – weighted graphs
Shortest paths using landmarks

Jaak Vilo
2020 Fall
Fast Fully Dynamic Landmark-based Estimation of Shortest Path Distances in Very Large Graphs

Konstantin Tretyakov  
University of Tartu  
Estonia  
kt@ut.ee

Abel Armas-Cervantes  
University of Tartu  
Estonia  
abel.armas@ut.ee

Luciano García-Bañuelos  
University of Tartu  
Estonia  
luciano.garcia@ut.ee

Jaak Vilo  
University of Tartu  
Estonia  
jaak.vilo@ut.ee

Marlon Dumas  
University of Tartu  
Estonia  
marlon.dumas@ut.ee

ABSTRACT
Computing the shortest path between a pair of vertices in a graph is a fundamental primitive in graph algorithms. Classical exact methods for this problem do not scale up to contemporary, rapidly evolving social networks with human interaction.

Keywords
Graph Databases, Shortest Paths, Social Networks, Landmarks, Treen, Dynamic Updates

ACM Conference on Information and Knowledge Management (CIKM) 2011
Fast Fully Dynamic Landmark-based Estimation of Shortest Path Distances in Very Large Graphs
Lisa kontakt

Lisa kontakt
Citsi Skype’i kasutajad ja lisad neid oma kontaktide hulka. Sesta nende Skype’i nimi, taisnimi või meiladress ja kõpsa Lea.

Mary

Sood lehele MySpace’i kontaktite. Sesta nende MySpace’i nimi ja kõpsa “Lea”.

Vali isik, keda otsisid ja vajuta ‘Lisa kontakt’:

<table>
<thead>
<tr>
<th>Taasimi</th>
<th>Skype’i nimi</th>
<th>Riikparkond</th>
<th>Profil</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syris, Mary</td>
<td>syris-mary</td>
<td>Venemaa, Moskva</td>
<td></td>
</tr>
<tr>
<td>Mary</td>
<td>abcmaries14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mary</td>
<td>affladorfer</td>
<td>Uuemaa, London</td>
<td></td>
</tr>
<tr>
<td>MARY</td>
<td>amore-feve-4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mary</td>
<td>beta, luz</td>
<td>Kanada, Calgary</td>
<td></td>
</tr>
<tr>
<td>Mary</td>
<td>bluestar88</td>
<td>Ameerika Ühendriigid, Parade</td>
<td></td>
</tr>
<tr>
<td>Mary</td>
<td>cpl67</td>
<td>TSM</td>
<td></td>
</tr>
<tr>
<td>Mary</td>
<td>ciociolotta2993</td>
<td>Itaalia, Rimini</td>
<td></td>
</tr>
<tr>
<td>Mary</td>
<td>forever-deep0247</td>
<td>Ameerika Ühendriigid, Indep</td>
<td></td>
</tr>
<tr>
<td>Mary</td>
<td>gg83, mary</td>
<td>Hina</td>
<td></td>
</tr>
</tbody>
</table>
Socially sensitive search

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

Landmark-based estimation

Mary Ann
Mary Lee
Landmark-based estimation

Basic Method

Mary Lee

Mary Ann

1 <= d <= 7

3 <= d <= 5

Landmark-based estimation
Shortest path tree

Least common ancestor
Least common ancestor

**Shortcutting**

Combining multiple landmarks
Combining multiple landmarks

Landmarks - BFS

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

Speed
Landmark-based approximation

Basic Method
- LCA
- Shortcutting
- Landmarks-BFS

Dynamic
Insertion of an edge
Deletion – more complicated
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>Graph file</th>
<th>Landmark file</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Basic</td>
<td>LCA/SC/LBFS</td>
</tr>
<tr>
<td>DBLP</td>
<td>27M</td>
<td>753K / 3.0M</td>
</tr>
<tr>
<td>Orkut</td>
<td>918M</td>
<td>3.0M / 12.0M</td>
</tr>
<tr>
<td>Twitter</td>
<td>9.3G</td>
<td>40M / 160M</td>
</tr>
<tr>
<td>Skype</td>
<td>27G</td>
<td>433M / 1.7G</td>
</tr>
</tbody>
</table>

---

per each landmark

---

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Method</th>
<th>No. of Landmarks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>Skype</td>
<td>Basic</td>
<td>0.18</td>
</tr>
<tr>
<td></td>
<td>LCA</td>
<td>1.06</td>
</tr>
<tr>
<td></td>
<td>SC</td>
<td>1.22</td>
</tr>
<tr>
<td></td>
<td>LBFS</td>
<td>5.10</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

- Highest degree
- Best coverage

Best Coverage

A  B  C  D  E  F

...
Best Coverage

A - u - v - w - B
C - v - x - D
E - y - z - s - F

...
Results

Error:

\[ d' - d \]

Landmark Selection
- Random
- Highest Degree
- Best Coverage

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)

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

### Summary

LCA | Shortcutting | Landmarks-BFS

Dynamic updates

Highest degree | Best coverage
Questions

LCA
Shortcutting
Landmarks-BFS
Dynamic updates
Highest degree
Best coverage

<table>
<thead>
<tr>
<th></th>
<th>Precomp.</th>
<th>Space</th>
<th>Query</th>
<th>Insertion</th>
<th>Deletion</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1+100x20min</td>
<td>170G</td>
<td>5ms / 16ms</td>
<td>0.030 ms</td>
<td>11ms</td>
<td>18% / 15%</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</td>
</tr>
<tr>
<td>Twitter</td>
<td>Basic</td>
<td>0.15</td>
</tr>
<tr>
<td></td>
<td>LCA</td>
<td>0.72</td>
</tr>
<tr>
<td></td>
<td>SC</td>
<td>0.82</td>
</tr>
<tr>
<td></td>
<td>LBFS</td>
<td><strong>240</strong></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.)
A community is a group of densely interconnected nodes
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 [35]. The administrator and the instructor are represented by nodes 1 and 33 respectively. Shaded squares represent individuals who ended up aligning with the club’s administrator after the fission of the club, open circles those who aligned with the instructor.
1. Divisive Methods

- Algorithm of Girvan-Newmann
  - 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.


Figure 1. Visualization of the steps of our algorithm. Each pass is made of two phases: one where modularity is optimized by allowing only local changes of communities, one where the found communities are aggregated in order to build a new network of communities. The passes are repeated iteratively until no increase of modularity is possible.
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

13.10.20

Family links

Friends

Colleagues

‘Family’ links

‘Friends’ links

‘Nerds & geeks’ links

Colleagues

‘Family’ links

‘Friends’ links
Link communities reveal multiscale complexity in networks

Yong-Yeol Ahn, James P. Gleeson & Sune Lehmann
References


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:

```
   c(s,v1)=16  c(v1,s)=0  c(v2,s)=0 ...
```

Conditions for Network Flow

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

- **Capacity Constraint:** \(\forall u,v \in V, \quad f(u,v) \leq c(u,v)\) (flow \(<\) 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:

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

The flow into the node is same as flow going out from the node and thus the flow is conserved. Also the total amount of flow from source $s = \text{total amount of flow into the sink t}$.

Example of a flow

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

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

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

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

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

Maximize \( v \)

Subject To

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

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

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:

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

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

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

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

Methods

Max-Flow Min-Cut Theorem

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

- Try to improve the flow, until we reach the maximum value of the flow
- The residual capacity of the network with a flow $f$ is given by:

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

\[
c_f(u,v) = c(u,v) - f(u,v)
\]

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
        let P be a path from s to t in G(x) do
            begin
                let P be a path from s to t in G(x);
                \[ \Delta := \delta(P); \]
                send \Delta units of flow along P;
                update the r's;
            end
    end  \{the flow x is now maximum\}.
```
Augmenting Paths (A Useful Concept)

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

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

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

The Ford-Fulkerson’s Algorithm

```latex
\text{FORDFULKERSON}(G, E, s, t)
\text{FOREACH } e \in E
\quad f(e) \leftarrow 0
\quad G_f \leftarrow \text{residual graph}
\text{WHILE} \ (\text{there exists augmenting path } P)
\quad f \leftarrow \text{augment}(f, P)
\quad \text{update } G_f
\text{ENDWHILE}
\text{RETURN } f
```

```latex
\text{AUGMENT}(f, P)
\quad b \leftarrow \text{bottleneck}(P)
\text{FOREACH } e \in P
\quad \text{IF } (e \in E)
\quad \quad \text{// backwards arc}
\quad \quad f(e) \leftarrow f(e) + b
\quad \text{ELSE}
\quad \quad \text{// forward arc}
\quad \quad f(e^b) \leftarrow f(e) - b
\text{RETURN } f
```
Proof of correctness of the algorithm

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

**Proof:** It's true at the beginning. Assume it's true after the first k-1 augmentations, and consider augmentation k along path P. The residual capacity $\Delta$ of P is the smallest residual capacity on P, which is integral. After updating, we modify the residual capacities by 0 or $\Delta$, and thus residual capacities stay integers.

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

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

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

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

---

**When is the flow optimal?**

A flow f is maximum flow in G if:

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

**Proof:**

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

2. Let $v=F_x(S,T)$ 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

This is the original network, and the original residual network.
Ford-Fulkerson Max Flow

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.
Ford-Fulkerson Max Flow

Find any s-t path

Determine the capacity $\Delta$ of the path.

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

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

Find any s-t path

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

Find any s-t path

Determine the capacity \( \Delta \) of the path.
Send \( \Delta \) units of flow in the path.
Update residual capacities.
Ford-Fulkerson Max Flow

There is no s-t path in the residual network. This flow is optimal.

Ford-Fulkerson Max Flow

These are the nodes that are reachable from node s.
Ford-Fulkerson Max Flow

Here is the optimal flow

Counterexample for termination

Ülesanne 54. Vaatleme voogu järgmisel joonisel.

Olgu $R = \sqrt{2} - 1$ (siis $R^m = R^{m+1} + R^{m+2}$) ja ülejäänud servadel suured lähilaskeväämed. Olgu esimene suurendav ahdel $s \rightarrow a \rightarrow d \rightarrow t$ ning järgnised suurendavad ahdel (tsüklis)

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

Näita, et Ford-Fulkersoni algoritm ei lõpeta tööd.
13.10.20

Distribution & Transportation

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

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

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...

Converting the Matching problem to Network Flow
Converting Matching to Network Flow

Converting Optimal Bipartite Matching to Network Flow