Introduction

- G=(V, E)
  - V = vertex set (nodes)
  - E = edge set (arcs)
- Graph representation
  - Adjacency list
  - Adjacency matrix
- Graph search
  - Breadth-first search (BFS)
  - Depth-first search (DFS)
  - Topological sort
  - Strongly connected components

Road Networks

In modeling a road network, the vertices may represent the cities or junctions, certain pairs of which are connected by roads/edges.
Electronic Circuits

In an electronic circuit, with junctions as vertices as components as edges.

A Simple Metabolic Pathway

Metabolic Regulation - Methionine Biosynthesis in E. coli

Evolutionary relationship among organisms bases on similarity of the primary sequences of their CYTOCHROME c protein

Green arrows - upregulation
Red arrows - downregulation
Thickness of arrow represents certainty of direction (up/down)
A complete graph

Filter
- choose a list of genes (MATING, marked in red)
- filter for these genes plus neighbouring genes from the graph

Mutation network $\Delta_{\text{mut}}$

Probability network $\Delta_{\text{mut}}$

unknown

ribos

mitochondial

aam

base

all

Green: small intestinal function; red: myometrial function; green: intestinal function; grey: unknown function.
Huge search spaces – graphs

- Almost any combinatorial search/optimisation problem can be described as a problem over a search space, described as an implicit graph.

Graphs

- Set of nodes $|V| = n$
- Set of edges $|E| = m$
  - Undirected edges/graph: pairs of nodes $(v,w)$
  - Directed edges/graph: pairs of nodes $(v,w)$
- Set of neighbors of $v$: set of nodes connected by an edge with $v$ (directed: in-neighbors, out-neighbors)
- Degree of a node: number of its neighbors (indegree, outdegree)
- Path: a sequence of nodes such that every two consecutive nodes constitute an edge
- Length of a path: number of nodes minus 1
- Distance between two nodes: the length of the shortest path between these nodes
- Diameter: the longest distance in the graph
Choose

- The boss wants to produce programs to solve the following two problems:
  - Euler circuit problem:
    - given a graph $G$, find a way to go through each edge exactly once.
  - Hamilton circuit problem:
    - given a graph $G$, find a way to go through each vertex exactly once.
- The two problems seem to be very similar.
- Person A takes the first problem and person B takes the second.
- Outcome: Person A quickly completes the program, whereas person B works 24 hours per day and is fired after a few months.

Euler Circuit: The original Konigsberg bridge 1735

"Can one walk across the seven bridges and never cross the same one twice?"
Euler circuit

- Every vertex of this graph has an even degree, therefore there exists an Eulerian graph. Following the edges in alphabetical order gives an Eulerian circuit/cycle.

Hamilton Circuit

Traveling salesman problem (TSP).

A joke (continued):

- **Why?** no body in the company has taken Algorithmics class.
- **Explanation:**
  - Euler circuit problem can be easily solved in polynomial time.
  - Hamilton circuit problem is proved to be \textbf{NP-hard}.
  - So far, no body in the world can give a polynomial time algorithm for a NP-hard problem.
  - Conjecture: there does not exist polynomial time algorithm for this problem.
Flavors of Graphs
The first step in any graph problem is determining which flavor of graph you are dealing with. Learning to talk the talk is an important part of walking the walk. The flavor of graph has a big impact on which algorithms are appropriate and efficient.

Directed vs. Undirected Graphs
A graph $G = (V, E)$ is undirected if edge $(x, y) \in E$ implies that $(y, x)$ is also in $E$.

Road networks between cities are typically undirected. Street networks within cities are almost always directed because of one-way streets. Most graphs of graph-theoretic interest are undirected.

Weighted vs. Unweighted Graphs
In weighted graphs, each edge (or vertex) of $G$ is assigned a numerical value, or weight.

The edges of a road network graph might be weighted with their length, drive-time or speed limit. In unweighted graphs, there is no cost distinction between various edges and vertices.

Simple vs. Non-simple Graphs
Certain types of edges complicate the task of working with graphs. A self-loop is an edge $(x, x)$ involving only one vertex. An edge $(x, y)$ is a multi-edge if it occurs more than once in the graph.

Any graph which avoids these structures is called simple.

Sparse vs. Dense Graphs
Graphs are sparse when only a small fraction of the possible number of vertex pairs actually have edges defined between them.

Graphs are usually sparse due to application-specific constraints. Road networks must be sparse because of road junctions. Typically dense graphs have a quadratic number of edges while sparse graphs are linear in size.

Cyclic vs. Acyclic Graphs
An acyclic graph does not contain any cycles. Trees are connected acyclic undirected graphs.

Directed acyclic graphs are called DAGs. They arise naturally in scheduling problems, where a directed edge $(x, y)$ indicates that $x$ must occur before $y$. 
**Implicit vs. Explicit Graphs**

Many graphs are not explicitly constructed and then traversed, but built as we use them.

A good example arises in backtrack search.

---

**Embedded vs. Topological Graphs**

A graph is **embedded** if the vertices and edges have been assigned geometric positions.

Example: TSP or Shortest path on points in the plane.
Example: Grid graphs.
Example: Planar graphs.

---

**Labeled vs. Unlabeled Graphs**

In *labeled* graphs, each vertex is assigned a unique name or identifier to distinguish it from all other vertices.

An important graph problem is **isomorphism testing**, determining whether the topological structure of two graphs are in fact identical if we ignore any labels.

---

**The Friendship Graph**

Consider a graph where the vertices are people, and there is an edge between two people if and only if they are friends.

This graph is well-defined on any set of people: SUNY SB, New York, or the world.
What questions might we ask about the friendship graph?

---

**Perception of/and experience**

- Simple example of 5 entities (persons) and their relationships
- Who would you prefer to be?
- Who wouldn’t you want to be?
- And what if the relationship means “company A sells to company B”?
- What if relationship means “love”?

---

**If I am your friend, does that mean you are my friend?**

A graph is **undirected** if \((x, y)\) implies \((y, x)\). Otherwise the graph is directed.

The “heard-of” graph is directed since countless famous people have never heard of me!

The “had-sex-with” graph is presumably undirected, since it requires a partner.
Am I my own friend?

An edge of the form \( (x, x) \) is said to be a loop.
If \( x \) is \( y \)'s friend several times over, that could be modeled using multiedges, multiple edges between the same pair of vertices.
A graph is said to be simple if it contains no loops and multiple edges.

Am I linked by some chain of friends to the President?

A path is a sequence of edges connecting two vertices. Since Mel Brooks is my father’s-sister’s-husband’s cousin, there is a path between me and him!

How close is my link to the President?

If I were trying to impress you with how tight I am with Mel Brooks, I would be much better off saying that Uncle Lenny knows him than to go into the details of how connected I am to Uncle Lenny.
Thus we are often interested in the shortest path between two nodes.

Is there a path of friends between any two people?

A graph is connected if there is a path between any two vertices.
A directed graph is strongly connected if there is a directed path between any two vertices.

Who has the most friends?

The degree of a vertex is the number of edges adjacent to it.
Complete graph

• Every node is connected to every other node

• Clique – fully connected subgraph of a graph

Enumerate all possible graphs with 5 nodes

Q: How many?

Subgraph

• Subset of vertices (m) \( V' \) is a subset of \( V \)

• Subset of edges (n) \( E' \) is a subset of \( E \), s.t. \( \{u,v\} \in E' \) if \( u,v \) both in \( V' \)

• \( 2^m \) different subsets of vertices (= many!)

Line graph

Given a graph \( G \), its line graph \( L(G) \) is a graph such that

• each vertex of \( L(G) \) represents an edge of \( G \); and

• two vertices of \( L(G) \) are adjacent if and only if their corresponding edges share a common endpoint ("are adjacent") in \( G \).

Representation For An Undirected Graph
Representation of Graphs

- **Adjacency list:** $O(V+E)$
  - Preferred for *sparse* graph
  - $|E| \ll |V|$ – Preferred for sparse graph
  - Adj[u] contains all the vertices $v$ such that there is an edge $(u,v) \in E$
  - Weighted graph: $w(u,v)$ is stored with vertex $v$ in Adj[u]
  - No quick way to determine if a given edge is present in the graph

- **Adjacency matrix:** $O(V^2)$
  - Preferred for *dense* graph
  - Symmetry for undirected graph
  - Weighted graph: store $w(u,v)$ in the $(u,v)$ entry
  - Easy to determine if a given edge is present in the graph

Tradeoffs Between Adjacency Lists and Adjacency Matrices

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space to store $(V+E)$ vertices</td>
<td>Adjacency list: $O(V+E)$</td>
</tr>
<tr>
<td>Time to find vertex $v$</td>
<td>Adjacency list: $O(1)$</td>
</tr>
<tr>
<td>Time to find edge $(u,v)$</td>
<td>Adjacency list: $O(1)$</td>
</tr>
<tr>
<td>Less memory for small graphs</td>
<td>Adjacency list: $O(V)$</td>
</tr>
<tr>
<td>Least memory for undirected</td>
<td>Adjacency list: $O(V)$</td>
</tr>
<tr>
<td>Graph is undirected</td>
<td>Adjacency list: $O(V)$</td>
</tr>
<tr>
<td>Graph is directed</td>
<td>Adjacency list: $O(V)$</td>
</tr>
<tr>
<td>Edge must be stored in Adj[u]</td>
<td>Adjacency list: $O(V)$</td>
</tr>
<tr>
<td>Edge must be stored in Adj[u]</td>
<td>Adjacency list: $O(V)$</td>
</tr>
<tr>
<td>Space to store problem</td>
<td>Adjacency list: $O(V)$</td>
</tr>
<tr>
<td>Space to store problem</td>
<td>Adjacency list: $O(V)$</td>
</tr>
</tbody>
</table>

Both representations are very useful and have different properties, although adjacency lists are probably better for most problems.

Edge list representation

- **AC**
- **AD**
- **CA**
- **CB**
- **DA**
- **DE**
- **EA**
- **EC**
- **ED**

Traversing a Graph

One of the most fundamental graph problems is to traverse every edge and vertex in a graph. For *efficiency*, we must make sure we visit each edge at most twice. For *correctness*, we must do the traversal in a systematic way so that we don’t miss anything. Since a maze is just a graph, such an algorithm must be powerful enough to enable us to get out of an arbitrary maze.
**Marking Vertices**

The key idea is that we must mark each vertex when we first visit it, and keep track of what have not yet completely explored. Each vertex will always be in one of the following three states:

- **undiscovered** – the vertex in its initial, virgin state.
- **discovered** – the vertex after we have encountered it, but before we have checked out all its incident edges.
- **processed** – the vertex after we have visited all its incident edges.

---

**Breadth-First Search (BFS)**

- Graph search: given a source vertex s, explores the edges of G to discover every vertex that is reachable from s
  - Compute the distance (smallest number of edges) from s to each reachable vertex
  - Produce a breadth-first tree with root s that contains all reachable vertices
  - Compute the shortest path from s to each reachable vertex
- BFS discovers all vertices at distance k from s before discovering any vertices at distance k+1

---

**Simple BFS from n**

```plaintext
enqueue(Q, n)
while Q not empty
    u = dequeue(Q)
    process u
    for each v in Adjacency(u) // neighbours
        if v not yet discovered
            enqueue(Q, v)
```

**Data Structure for BFS**

- Adjacency list
- color[u] for each vertex — WHITE if u has not been discovered — BLACK if u and all its adjacent vertices have been discovered — GRAY if u has been discovered, but has some adjacent white vertices
  - Frontier between discovered and undiscovered vertices
- d[u] for the distance from (source) s to u
- π(u) for predecessor of u
- FIFO queue Q to manage the set of gray vertices — Q stores all the gray vertices

---

**BFS(G, s)**

```plaintext
1 for each vertex u ∈ V(G) – {s}
2   do color[u] ← WHITE
3     d[u] ← ∞
4     π[u] ← NIL

5 color[s] ← GRAY
6 d[s] ← 0
7 π[s] ← NIL
8 Q ← Ø
9 ENQUEUE(Q, s)
10 while Q ≠ Ø
11   do u ← DEQUEUE(Q)
12     for each v ∈ Adj[u]
13         do if color[v] = WHITE
14             then color[v] ← GRAY
15             d[v] ← d[u] + 1
16             π[v] ← u
17             ENQUEUE(Q, v)
18     color[u] ← BLACK
```

---

**Initialization**

Set up s and initialize Q

Explore all the vertices adjacent to u and update d, π and Q
Analysis of BFS

- $O(|V|+|E|) = O(n+m)$
  - Each vertex is enqueued ($O(1)$) at most once $\Rightarrow O(n)$
    - No vertex is re-painted white
      - A vertex is inserted into queue and retrieved from there only once
  - Each adjacency list is scanned at most once $\Rightarrow O(m)$
Shortest path

• Print out the vertices on a shortest path from s to v

BFS – tree edges, cross edges

• For undirected graph G, in BFS, every edge will be
  – Tree edge, or
  – Cross-edge (not ancestor, not descendent)

Connected Components

The connected components of an undirected graph are the separate “pieces” of the graph such that there is no connection between the pieces.

Many seemingly complicated problems reduce to finding or counting connected components. For example, testing whether a puzzle such as Rubik’s cube or the 15-puzzle can be solved from any position is really asking whether the graph of legal configurations is connected.

Anything we discover during a BFS must be part of the same connected component. We then repeat the search from any undiscovered vertex (if one exists) to define the next component, until all vertices have been found.

Rubik’s cube

There are many algorithms to solve scrambled Rubik’s Cubes. The maximum number of face turns needed to solve any instance of the Rubik’s Cube is 20. [1] and the maximum number of quarter turns is 26. [2] These numbers are also the diameters of the corresponding Cayley graphs of the Rubik’s Cube group. An algorithm that solves a cube in the minimum number of moves is known as God’s algorithm.

In 2009, Tomas Rokicki proved that 29 moves in the quarter-turn metric is enough to solve any scrambled cube. [19] And in 2014, Tomas Rokicki and Morley Davidson proved that the maximum number of quarter-turns needed to solve the cube is 26. [20]


Two-Coloring Graphs

The vertex coloring problem seeks to assign a label (or color) to each vertex of a graph such that no edge links any two vertices of the same color.

A graph is bipartite if it can be colored without conflicts while using only two colors. Bipartite graphs are important because they arise naturally in many applications.

For example, consider the “had-sex-with” graph in a heterosexual world. Men have sex only with women, and vice versa. Thus gender defines a legal two-coloring.
Bipartite graphs

- people and groups
- men-women
- Stable marriage
  - find matching that will not be “broken” by inevitable divorces
- Apples and Oranges

Finding a Two-Coloring

We can augment breadth-first search so that whenever we discover a new vertex, we color it the opposite of its parent.

```java
recursive(graph G) {
  for (v : G) {
    color[v] = UNCOLORED;
  }
  if (biparta == TRUE) {
    initialize source(u);
    for (v : G) {
      if (discovered[v] == (1 - biparta) && color[v] == WHITE) {
        color[v] = BLACK;
      }
    }
  }
}
```

Depth-First Search (DFS)

- DFS: search deeper in the graph whenever possible
- Edges are explored out of the most recently discovered vertex v that still has unexplored edges leaving it
- When all of v’s edges have been explored (finished), the search backtracks to explore edges leaving the vertex from which v was discovered
- This process continues until we have discovered all the vertices that are reachable from the original source vertex
- If any undiscovered vertices remain, then one of them is selected as a new source and the search is repeated from that source
- The entire process is repeated until all vertices are discovered
- DFS will create a forest of DFS-trees

Recursive:

DFS( u ):
  process u
  for each v in Adjacency( u )
    DFS( v ) if v undiscovered

Data Structure for DFS

- Adjacency list
- color[u] for each vertex
  - WHITE if u has not been discovered
  - GRAY if u is discovered but not finished
  - BLACK if u is finished
- Timestamps: 1 ≤ d[u] < f[u] ≤ 2|V|
  - d[u] records when u is first discovered (and grayed)
  - f[u] records when the search finishes examining u’s adjacency list (and blacken u)
- π[u] for predecessor of u
**The Key Idea with DFS**

A depth-first search of a graph organizes the edges of the graph in a precise way. In a DFS of an undirected graph, we assign a direction to each edge, from the vertex which discover it:

**DFS: initialise and visit all yet unexplored vertices**

```
DFS(G)
1 for each vertex \( u \in V[G] \)
2 \[ \text{do } \text{color}[u] \leftarrow \text{WHITE} \]
3 \[ \pi[u] \leftarrow \text{NIL} \]
4 \[ \text{time} \leftarrow 0 \]
5 for each vertex \( u \in V[G] \)
6 \[ \text{do if color}[u] = \text{WHITE} \]
7 \[ \text{then DFS-visit}(u) \]
```

**DFS-visit – visit all reachable nodes**

```
DFS-visit(u)
1 \[ \text{color}[u] \leftarrow \text{GRAY} \quad \text{▷ White vertex } u \text{ has just been discovered.} \]
2 \[ \text{time} \leftarrow \text{time} + 1 \]
3 \[ \pi[u] \leftarrow \text{time} \]
4 for each \( v \in \text{Adj}[u] \) \quad \text{▷ Explore edge } (u, v).
5 \[ \text{do if color}[v] = \text{WHITE} \]
6 \[ \text{then } \pi[v] \leftarrow u \]
7 \[ \text{DFS-visit}(v) \]
8 \[ \text{color}[u] \leftarrow \text{BLACK} \quad \text{▷ Blacken } u; \text{ it is finished.} \]
9 \[ f[u] \leftarrow \text{time} \leftarrow \text{time} + 1 \]
```

**Example (DFS)**
(Courtesy of Prof. Jim Anderson)

```
Example (DFS)
```

```
Example (DFS)
```

```
Example (DFS)
```
Properties of DFS

- Time complexity: $\Theta(V+E)$
  - Loops on lines 1-3 and 5-7 of DFS: $\Theta(V)$
  - DFS-VISIT
    - Called exactly once for each vertex
    - Loops on lines 4-7 for a vertex $v$: $|\text{Adj}[v]|$
  - Total time = $\sum_{v \in V} |\text{Adj}[v]| = \Theta(E)$

- DFS results in a forest of trees
- Discovery and finishing times have parenthesis structure

Depth-First Search

DFS has a neat recursive implementation which eliminates the need to explicitly use a stack. Discovery and final times are a convenience to maintain.

Another Example of DFS
Parenthesis Theorem

Theorem 22.7
For all \(u, v\), exactly one of the following holds:
1. \(d[u] < f[u] < d[v] < f[v]\) or \(d[v] < f[v] < d[u] < f[u]\) and neither \(u\) nor \(v\) is a descendant of the other.
2. \(d[u] < d[v] < f[v] < f[u]\) and \(v\) is a descendant of \(u\).
3. \(d[v] < d[u] < f[u] < f[v]\) and \(u\) is a descendant of \(v\).

Corollary
\(v\) is a proper descendant of \(u\) if and only if \(d[u] < d[v] < f[v] < f[u]\).

Proof
We begin with the case in which \(d[u] < d[v]\).
• There are two subcases to consider, according to whether \(d[v] < f[u]\) or not.
• The first subcase occurs when \(d[v] < f[u]\), so \(v\) was discovered while \(u\) was still gray. This implies that \(v\) is a descendant of \(u\). Moreover, since \(v\) was discovered more recently than \(u\), all of its outgoing edges are explored, and \(v\) is finished, before the search returns to and finishes \(u\). In this case, therefore, the interval \([d[v], f[v]]\) is entirely contained within the interval \([d[u], f[u]]\).
• In the other subcase, \(f[u] < d[v]\), and inequality (22.2) implies that the intervals \([d[u], f[u]]\) and \([d[v], f[v]]\) are disjoint.
• Because the intervals are disjoint, neither vertex was discovered while the other was gray, and so neither vertex is a descendant of the other.
• The case in which \(d[v] < d[u]\) is similar, with the roles of \(u\) and \(v\) reversed in the above argument.

Parenthesis theorem
In any depth-first search of a (directed or undirected) graph \(G = (V, E)\), for any two vertices \(u\) and \(v\), exactly one of the following three conditions holds:
1. the intervals \([d[u], f[u]]\) and \([d[v], f[v]]\) are entirely disjoint, and neither \(u\) nor \(v\) is a descendant of the other in the depth-first forest,
2. the interval \([d[u], f[u]]\) is contained entirely within the interval \([d[v], f[v]]\), and \(u\) is a descendant of \(v\) in a depth-first tree, or
3. the interval \([d[v], f[v]]\) is contained entirely within the interval \([d[u], f[u]]\), and \(v\) is a descendant of \(u\) in a depth-first tree.

Depth-First Trees
• Predecessor subgraph defined slightly different from that of BFS.
• The predecessor subgraph of DFS is \(G_p = (V, E_p)\) where \(E_p = \{(\pi[v], v) : v \in V\text{ and } \pi[v] \neq \text{NIL}\}\).
  -- How does it differ from that of BFS?
  -- The predecessor subgraph \(G_p\) forms a depth-first forest composed of several depth-first trees. The edges in \(E_p\) are called tree edges.

Definition:
Forest: An acyclic graph \(G\) that may be disconnected.
White-path Theorem

**Theorem 22.9**
\( u \) is a descendant of \( v \) if and only if at time \( d[u] \), there is a path \( u \rightarrow v \) consisting of only white vertices. (Except for \( u \), which was just colored gray.)

Classification of Edges

- **Tree edge**: in the depth-first forest. Found by exploring \((u, v)\). -- \( v \) was white.
- **Back edge**: \((u, v)\), where \( u \) is a descendant of \( v \) (in the depth-first tree). -- \( v \) was gray.
- **Forward edge**: \((u, v)\), where \( v \) is a descendant of \( u \), but not a tree edge. -- \( v \) was black and \( d(u) < d(v) \).
- **Cross edge**: any other edge. Can go between vertices in same depth-first tree or in different depth-first trees. -- \( v \) was black and \( d(u) > d(v) \).

---

Types of edges in DFS

1. **Tree edge**
2. **Back edge**
3. **Forward edge**
4. **Cross edge**

Types of edges in DFS

- **Tree edge**
- **Back edge**
- **Forward edge**
- **Cross edge**

---

Edge Classification for DFS

Every edge is either:

1. A Tree Edge
2. A Back Edge
3. A Forward Edge
to a descendant
4. A Cross Edge
to a descendant

On any particular DFS or BFS of a directed or undirected graph, each edge gets classified as one of the above.

---

DFS: Tree Edges and Back Edges Only

The reason DFS is so important is that it defines a very nice ordering to the edges of the graph.

In a DFS of an undirected graph, every edge is either a tree edge or a back edge.

Why? Suppose we have a forward edge. We would have encountered \([4, 1]\) when expanding 4, so this is a back edge.

(for undirected graphs)
(for undirected graphs)

No Cross Edges in DFS

Suppose we have a cross-edge

\[ \text{DFS Application: Finding Cycles} \]

Back edges are the key to finding a cycle in an undirected graph. Any back edge going from \( x \) to an ancestor \( y \) with the path in the tree from \( y \) to \( x \).

Suppose that \( G \) contains a cycle \( c \). We show that a DFS of \( G \) yields a back edge. Let \( v \) be the first vertex to be discovered in \( c \), and let \( (u, v) \) be the preceding edge in \( c \). At time \( d[v] \), the vertices of \( c \) form a path of white vertices from \( v \) to \( u \). By the white-path theorem (Theorem 22.9), vertex \( u \) becomes a descendant of \( v \) in the depth-first forest. Therefore, \( (u, v) \) is a back edge.

Lemma – DAG acyclicity

- DAG is acyclic if and only if DFS of \( G \) yields no back edges

Suppose that there is a back edge \( (u, v) \). Then vertex \( v \) is an ancestor of vertex \( u \) in the depth-first forest. There is thus a path from \( v \) to \( u \) in \( G \), and the back edge \( (u, v) \) completes a cycle.

Suppose that \( G \) contains a cycle \( c \). We show that a DFS of \( G \) yields a back edge. Let \( v \) be the first vertex to be discovered in \( c \), and let \( (u, v) \) be the preceding edge in \( c \). At time \( d[v] \), the vertices of \( c \) form a path of white vertices from \( v \) to \( u \). By the white-path theorem (Theorem 22.9), vertex \( u \) becomes a descendant of \( v \) in the depth-first forest. Therefore, \( (u, v) \) is a back edge.

Articulation Vertices

Suppose you are a terrorist, seeking to disrupt the telephone network. Which station do you blow up?

An articulation vertex is a vertex of a connected graph whose deletion disconnects the graph.

Clearly connectivity is an important concern in the design of any network.

Articulation vertices can be found in \( O(n(n+m)) \) – just delete each vertex to do a DFS on the remaining graph to see if it is connected.

A Faster \( O(n + m) \) DFS Algorithm

In a DFS tree, a vertex \( v \) (other than the root) is an articulation vertex iff \( v \) is not a leaf and some subtree of \( v \) has no back edge incident until a proper ancestor of \( v \).

Problems 22.9: Articulation points, bridges, and biconnected components

Let \( G = (V, E) \) be a connected, undirected graph. An articulation point of \( G \) is a vertex whose removal disconnects it. A bridge of \( G \) is an edge whose removal disconnects it. A biconnected component of \( G \) is a maximal set of edges such that any two edges in the set form a common simple cycle. Figure 22.10 illustrates these definitions. We can determine the articulation points, bridges, and biconnected components using depth-first search. Let \( G'_v = (V, E'_v) \) be a depth-first tree of \( G \).

- \( G'_v \) is a depth-first tree of \( G \).
- The articulation points, bridges, and biconnected components of a connected, undirected graph for use in Problem 22.9. The articulation points are the heavy, shaded vertices, the bridges are the heavily shaded edges, and the biconnected components are the edges in the shaded region, with a box number as shown.
- b. Let \( \text{v} \) be a nonarticulation vertex of \( G \). Compute that is an articulation point of \( G \) and only if it has at least one child in \( G \).
- a. Prove that the root of \( G \) is an articulation point of \( G \).
- b. Let \( \text{v} \) be a nonarticulation vertex of \( G \). Compute that is an articulation point of \( G \) and only if it has at least one child in \( G \).
- c. Let \( \text{v} \) be a nonarticulation vertex of \( G \). Compute that is an articulation point of \( G \) and only if it has at least one child in \( G \).
Recursive:

\[
\text{DFS}(u) : \\
\text{process } u \\
\text{for each } v \text{ in } \text{Adjacency}(u) \\
\text{DFS}(v) \text{ if } v \text{ undiscovered}
\]

Simple DFS from n (non-recursive)

\[
\text{push } (Q, n) \\
\text{while } u = \text{pop}(Q) \\
\text{process } u \\
\text{for each } v \text{ in reverse } \text{Adjacency}(u) \\
\text{push } (Q, v)
\]

DFS
- Stack

BFS
- Queue

Randomised search
Use:
Randomized Queue
Topological Sort

- A topological sort of a directed acyclic graph (DAG) is a linear order of all its vertices such that if \( G \) contains an edge \((u,v)\), then \( u \) appears before \( v \) in the ordering.
  - If the graph contains cycles, no linear ordering is possible.
  - A topological sort can be viewed as an ordering of its vertices along a horizontal line so that all directed edges go from left to right.
- DAG are used in many applications to indicate precedence among events.
Topological Sort

- $O(V+E)$

**TOPOLOGICAL-SORT(G)**

1. call DFS(G) to compute finishing times $f[v]$ for each vertex $v$
2. as each vertex is finished, insert it onto the front of a linked list
3. return the linked list of vertices

Example

(Courtesy of Prof. Jim Anderson)

Linked List:

Example

Linked List:

Example

Linked List:

Example

Linked List:
Example

Linked List:

A → B → D → C → E

Example

Linked List:

A → B → D → C → E

Example

Linked List:

A → B → D → C → E

Example

Linked List:

A → B → D → C → E

Example

Linked List:

A → B → D → C → E

Example

Linked List:

A → B → D → C → E
Theorem: Correctness of Topological sort

- TOPOLOGICAL-SORT(G) produces a topological sort of a directed acyclic graph G.
  - Suppose that DFS is run on a given DAG G to determine finishing times for its vertices. It suffices to show that for any pair of distinct vertices u, v, if there is an edge in G from u to v, then f[v] < f[u].
  - The linear ordering is corresponding to finishing time ordering
  - Consider any edge (u, v) explored by DFS(G). When this edge is explored, v cannot be gray (otherwise, (u, v) will be a back edge). Therefore v must be either white or black.
    - If v is white, v becomes a descendant of u, f[v] < f[u] (ex. pants & shoes)
    - If v is black, it has already been finished, so that f[v] has already been set f[v] < f[u] (ex. belt & jacket)

STRONGLY CONNECTED COMPONENTS

Strongly Connected Components
A directed graph is strongly connected if there is a directed path between any two vertices.
The strongly connected components of a graph is a partition of the vertices into subsets (maximal) such that each subset is strongly connected.

Observe that no vertex can be in two maximal components, so it is a partition.

Component Graph
- \( G^{SCC} = (V^{SCC}, E^{SCC}) \).
- \( V^{SCC} \) has one vertex for each SCC in G.
- \( E^{SCC} \) has an edge if there’s an edge between the corresponding SCC’s in G.
- \( G^{SCC} \) for the example considered:
Lemma 22.13
Let C and C’ be distinct SCC’s in G, let u, v ∈ C, u’, v’ ∈ C’, and suppose there is a path u → u’ in G. Then there cannot also be a path v → v’ in G.

Proof:
• Suppose there is a path v → v’ in G.
• Then there are paths u → u’ → v’ → v and v → v’ → u → u’ in G.
• Therefore, u and v’ are reachable from each other, so they are not in separate SCC’s.

Transverse of a Directed Graph
• \( G^T = \text{transpose} \) of directed G.
  \[ G^T = (V, E^T), E^T = \{ (v, u) : (u, v) \in E \} \]
  • \( G^T \) is G with all edges reversed.
• Can create \( G^T \) in \( \Theta(V + E) \) time if using adjacency lists.
• \( G \) and \( G^T \) have the same SCC’s. (u and v are reachable from each other in \( G \) if and only if reachable from each other in \( G^T \).)

Algorithm to determine SCCs

1. call DFS(G) to compute finishing times \( f[u] \) for all \( u \)
2. compute \( G^T \)
3. call DFS(\( G^T \)), but in the main loop, consider vertices in order of decreasing \( f[u] \) (as computed in first DFS)
4. output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC

Time: \( \Theta(V + E) \).

Example: On board.

Kosaraju’s algorithm, 1978
Tarjan – 1972
Gabow - 1990 (Cheriyan, Melhorn 1996)
**Example**

Let $G$ be a graph with vertices $V$ and edges $E$. Suppose there is an edge $(u, v) \in E$ such that $u \in C$ and $v \in C'$, then $f(C) > f(C')$.

**Proof:**

- Case 1: $d(C) > d(C')$
  - Let $x$ be the first vertex discovered in $C$.
  - At time $d(x)$, all vertices in $C$ and $C'$ are white. Thus, there exist paths of white vertices from $x$ to all vertices in $C$ and $C'$.
  - By the white-path theorem, all vertices in $C$ and $C'$ are descendants of $x$ in the depth-first tree.
  - By the parenthesis theorem, $f(x) = f(C) > f(C')$.

- Case 2: $d(C) > d(C')$
  - Let $y$ be the first vertex discovered in $C$.
  - At time $d(y)$, all vertices in $C$ are white and there is a white path from $y$ to each vertex in $C'$.
  - By the white-path theorem, all vertices in $C$ and $C'$ are descendants of $y$.
  - By the parenthesis theorem, $f(y) = f(C) > f(C')$.

**SCCs and DFS finishing times**

**Lemma 22.14**

Let $C$ and $C'$ be distinct SCC's in $G = (V, E)$. Suppose there is an edge $(u, v) \in E$ such that $u \in C$ and $v \in C'$. Then $f(C) > f(C')$.

**Correctness of SCC**

- When we do the second DFS, on $G'$, start with SCC $C$ such that $f(C)$ is maximum.
  - The second DFS starts from some $x \in C$, and it visits all vertices in $C$.
  - Corollary 22.15 says that since $f(C) > f(C')$ for all $C \neq C'$, there are no edges from $C$ to $C'$ in $G'$.
  - Therefore, DFS will visit only vertices in $C$.
  - Which means that the depth-first tree rooted at $x$ contains exactly the vertices of $C$.

**How does it work?**

- **Idea:**
  - By considering vertices in second DFS in decreasing order of finishing times from first DFS, we are visiting vertices of the component graph in topologically sorted order.
  - Because we are running DFS on $G'$, we will not be visiting any $v$ from a $u$, where $v$ and $u$ are in different components.

- **Notation:**
  - $d[u]$ and $f[u]$ always refer to first DFS.
  - Extend notation for $d$ and $f$ to sets of vertices $U \subseteq V$:
    - $d(U) = \min_{u \in U} \{ d[u] \}$ (earliest discovery time)
    - $f(U) = \max_{u \in U} \{ f[u] \}$ (latest finishing time)

**SCCs and DFS finishing times**

**Lemma 22.15**

Let $C$ and $C'$ be distinct SCC's in $G = (V, E)$. Suppose there is an edge $(u, v) \in E$ such that $u \in C$ and $v \in C'$. Then $f(C) > f(C')$.

**Corollary 22.15**

Let $C$ and $C'$ be distinct SCC's in $G = (V, E)$. Suppose there is an edge $(u, v) \in E$, where $u \in C$ and $v \in C'$, then $f(C') > f(C)$.

**Proof:**

- $(u, v) \in E' \Rightarrow (v, u) \in E$.
- Since SCC's of $G$ and $G'$ are the same, $f(C') > f(C)$, by Lemma 22.14.
Correctness of SCC

- The next root chosen in the second DFS is in SCC $C'$ such that $f(C')$ is maximum over all SCC’s other than $C$.
  - DFS visits all vertices in $C'$, but the only edges out of $C'$ go to $C$, which we’ve already visited.
  - Therefore, the only tree edges will be to vertices in $C$.
- We can continue the process.
- Each time we choose a root for the second DFS, it can reach only
  - vertices in its SCC—get tree edges to these,
  - vertices in SCC’s already visited in second DFS—get no tree edges to these.

Strongly Connected Components Example

Why does strongly connected component method work?

- See CLRS (2-3 pages)

Advanced Algorithmics (6EAP) Graphs II

Jaak Vilo
2011 Spring

Weighted Graph Algorithms

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

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

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

- May have more than one MST with the same weight.

- Two classic algorithms: \(O(E \lg V)\) - 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?

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**

\[
\begin{align*}
1 & \quad A \leftarrow \emptyset \\
2 & \quad \text{while } A \text{ does not form a spanning tree} \\
3 & \quad \text{do find an edge } (u, v) \text{ that is safe for } A \\
4 & \quad A \leftarrow A \cup \{(u, v)\} \\
5 & \quad \text{return } A
\end{align*}
\]

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

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

---

Proof of Theorem 23.1

- Let $T$ be a MST that includes $A$, and assume $T$ does not contain the light edge $(u, v)$, since if it does, we are done.
- Construct another MST $T'$ that includes $A \cup \{(u, v)\}$ from $T$
  
  – Next slide
  
  – $T' = T - \{(x, y)\} \cup \{(u, v)\}$
  
  – $T'$ 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 $(x, y) \notin A \Rightarrow A \subseteq T'$
    
    – $A \cup \{(u, v)\} \subseteq T'$

---

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 $A$
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, \( \text{fringe} \) (meaning there exists an edge from a tree vertex to \( v \)) or \( \text{unseen} \) (meaning \( v \) is more than one edge away).

\[
\text{Prim-MST}(G)
\]
  Select an arbitrary vertex \( r \) to start the tree from.
  While (there are still non-tree vertices)
    Select the edge of minimum weight between a tree and node
    Add the selected edge and vertex to the tree \( T_{\text{tree}} \).

This creates a spanning tree, since no cycle can be introduced, but is it minimum?

Prim's Algorithm in Action

[Diagrams showing the construction of the minimum spanning tree using Prim's Algorithm]
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

MST-PRIM(G, w, r)

1. for each u ∈ V
   2. do D[u] ← ¥
   3. 0[f] ← 0
   4. MH ← make-heap(D, 0)
   5. r ← r
6. while MH ≠ ∅ do
7.     (u, e) ← MH.extractMin()
8.     add (u, e) to T
9.     for each v ∈ Adjacent (u)
10.     do if v ∈ MH and w(u, v) < D[v]
11.        then D[v] ← w(u, v)
12.        MH.decreaseDistance(D[v], v, (u, v))
13. return T // T is a MST

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

Illustration of MST-PRIM
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: $O(\log V)$
  - The for loop in lines 8-11 is executed $O(E)$ times altogether
    - Line 11: DECREASE-KEY operation: $O(\log V)$
  - Total performance = $O(V\log V + E\log V) = O(E\log V)$
- Use Fibonacci heap to implement the min-priority queue $Q$
  - $O(E + V\log V)$

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.

Fast Kruskal Implementation

Put the edges in a heap

```
count = 0
while (count < n - 1) do
  get next edge $(v, w)$
  if (component(v) $\neq$ 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 $O(\log n)$ better than $O(m \log m)$?

Kruskal algorithm for MST

```
KRUSKAL(G):
1 A = \emptyset
2 foreach v in G.V:
3   MAKE-SET(v)
4 foreach (u, v) ordered by weight(u, v), increasing:
5   if FIND-SET(u) $\neq$ FIND-SET(v):
6     A = A $\cup$ {(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 $\langle x, y \rangle$ Kruskal adds such that the set of edges cannot be extended into a minimum spanning tree. When we added $\langle x, y \rangle$ there previously was no path between $x$ and $y$, or it would have created a cycle. Thus if we add $\langle x, y \rangle$ to the optimal tree it must create a cycle. At least one edge in this cycle must have been added after $\langle x, y \rangle$, so it must have a heavier weight. Deleting this heavy edge leave a better MST than the optimal tree? A contradiction!

How fast is Kruskal’s algorithm?

What is the simplest implementation?

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

The total time is $O(mn)$, but can we do better?

Fast Component Tests Give Fast MST

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

- **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 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($j$) now equals find($i$).

See the lecture on trees...

Problem of the Day

Suppose we are given the minimum spanning tree $T$ of a given graph $G$ (with $n$ vertices and $m$ edges) and a new edge $e = \langle u, v \rangle$ of weight $w(e)$ 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.
**SINGLE-SOURCE SHORTEST PATHS**  
(CHAPTER 24)

**Table 20.1 Cost of MST algorithms**

<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>$E \log V$</td>
<td>conservative upper bound</td>
</tr>
<tr>
<td>Prim (PFS, d-heap)</td>
<td>$E \log E$</td>
<td>linear unless extremely sparse</td>
</tr>
<tr>
<td>Kruskal</td>
<td>$E \log V$</td>
<td>sort cost dominates</td>
</tr>
<tr>
<td>Kruskal (priority sort)</td>
<td>$E + X \log V$</td>
<td>cost depends on longer edge</td>
</tr>
<tr>
<td>Boruvka</td>
<td>$E \log V$</td>
<td>conservative upper bound</td>
</tr>
</tbody>
</table>

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 20.60). 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.

**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,

---

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:

\[
    w(p) = \sum_{i=0}^{k} w_{v_i, v_{i+1}}
\]

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

\[
    d(u, v) = \begin{cases} 
    \min\{w(p): u \rightarrow v\} & \text{if there is a path from $u$ to $v$;} \\
    \infty & \text{otherwise.}
\end{cases}
\]

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

---

**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. ‘hiveme’ 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.
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 \( p = \langle v_1, v_2, ..., v_k \rangle \) be a shortest path from vertex \( v_1 \) to \( v_k \), and for any \( i \) and \( j \) such that \( 1 \leq i \leq j \leq k \), let \( p_{ij} = \langle v_{1i}, v_2, ..., v_{ji} \rangle \) be the subpath of \( p \) from vertex \( v_i \) to \( v_j \). Then \( p_{ij} \) is a shortest path from vertex \( v_i \) to \( v_j \).

---

Negative-Weight Edges and Cycles

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

---

Relaxation

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

\[
\text{Initialze-Single-Source}(G, s)
\]

1. for each vertex \( v \in V[G] \)
2. \( d[v] \leftarrow \infty \)
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 \(p[v]\).

\[
\text{Relax}(u, v, w) \quad \begin{cases} 
1 & \text{if } d[v] > d[u] + w(u, v) \\
2 & \text{then } d[v] = d[u] + w(u, v) \\
3 & p[v] \leftarrow u 
\end{cases}
\]

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. \( \text{RELAX}(u, v, w) \)
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 \(\text{distTo}\) are those leaving a vertex whose \(\text{distTo}\) value changed in the previous pass. To keep track of such vertices, we use a FIFO queue. The operation of the algorithm for our
Shortest paths on a DAG

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

O(V + E)

Dijkstra’s Algorithm (1956,1959)

- Solve the single-source shortest-paths problem on a weighted, directed graph when all edge weights are nonnegative
- Data structure
  - S: a set of vertices whose final shortest-path weights have already been determined
  - Q: a min-priority queue keyed by their d values
- Idea
  - Repeatedly select the vertex u ∈ V - S (kept in Q) with the minimum shortest-path estimate, add s 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. \( S \leftarrow \emptyset \)
3. \( Q \leftarrow V[G] \)
4. \textbf{while} \( Q \neq \emptyset \)
   5. \( u \leftarrow \text{EXTRACT-MIN}(Q) \)
   6. \( S \leftarrow S \cup \{u\} \)
   7. \textbf{for each vertex} \( v \in \text{Adj}[u] \)
      8. \( \text{RELAX}(u, v, w) \)

\textbf{Note:} relax requires updating of min values in \( Q \)

\[ \text{Figure 24A} \] The execution of Dijkstra's algorithm. The source \( s \) is the leftmost vertex. The shortest-path estimates are shown by the vertices, and shaded edges indicate predecessor values. Black vertices are in the set \( S \), and white vertices are in the non-processed queue \( Q \). (a) The situation just before the first iteration of the \textit{while} loop of lines 4-5. The shaded vertex has the minimum of values and is chosen as vertex \( u \) in line 5. (b) The situation after each successive iteration of the \textit{while} loop. The shaded vertex in each part is chosen as vertex \( u \) in line 5 of the next iteration. The \( u \) and \( w \) values shown in part (f) are the final values.

\[ \text{Wikipedia - animation} \]
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 \(O(V)\)
  - Line 7-8: for loop and relaxation \(O(E)\)
- 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

Examples of shortest paths depending on start node

Eucl- Path

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

- In applications where networks model maps, our primary interest is often in finding the best route from one place to another. In this section, we examine a strategy for this problem: a fast algorithm for the source-sink shortest-path problem in Euclidean networks, which are networks whose vertices are points in the plane and whose edge weights are defined by the geometric distances between the points.
- These networks satisfy two important properties that do not necessarily hold for general edge weights. First, the distances satisfy the triangle inequality: The distance from s to d is never greater than the distance from s to x plus the distance from x to d. Second, vertex positions give a lower bound on path length: No path from s to d will be shorter than the distance from s to d. The algorithm for the source-sink shortest paths problem that we examine in this section takes advantage of these two properties to improve performance.

Calculating paths by matrix operations

Paths of length 2

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

Paths of length 2

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

Paths of length 2

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

Paths of length 2

<table>
<thead>
<tr>
<th>0 1 2 3 4 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 1 0</td>
</tr>
<tr>
<td>1 1 0 0 0 0</td>
</tr>
<tr>
<td>2 0 1 0 0 0</td>
</tr>
<tr>
<td>3 0 0 1 0 1</td>
</tr>
<tr>
<td>4 0 0 0 0 1</td>
</tr>
<tr>
<td>5 0 0 0 0 1</td>
</tr>
</tbody>
</table>
\[ c_{ij} = \sum_{k} a_{ik} b_{kj} \]

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

**Binary matrix “multiplication”**

\[ a+b \to a \text{ or } b \quad a*b \to a \text{ and } b \]

```
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**

The textbook algorithm for computing the product of two V-by-V matrices computes, for each \( a \) and \( i \), the dot product of row \( a \) in the first matrix and row \( i \) in the second matrix, as follows:

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

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

```java
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;
```
Diagonal 0 or 1

Transitive closure

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

Transitive closure

G * G * G * ...

G[i][j] and G[j][k] => G[i][k]

1 on diagonal - link to itself

Figure 14-1 Transitive closure

This digraph has no right directed edges, but its transitive closure does. Shown are the right directed paths connecting them. The property that the transitive closure of direction is reflected in the transitive closure of direction. For example, lines 6, 7, and 8 in the adjacency matrix for the transitive closure are identical, as are columns 6, 7, and 8 because these vertices are on a directed cycle in the digraph.

Book: Sedgewick, Algorithms ...

- 19.3. Reachability and Transitive Closure
- Complexity...
- for i=1 to |V| do V[i] = V[i-1]+V
- V^3 operations for V^2, V^3, ... V^v
- = O(V^3)
- Use exponential: 2 => 4 => 8 => 16 ... steps.
- V^2*V^2=V^4, V^3*V^3=V^6, ... => O(\log V) * V^3
- Can we avoid so many cycles?
Multiply:

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

Transitive closure:

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

Proof

• Proof: transitive closure by induction on i.
  • Iteration 1: either s=t or the path s-0-t.
  • It 2: all the paths between s and t that include 1 and perhaps 0, such as s-1-t, s-1-0-t, and s-0-1-t.
  • Inductive hypothesis: The 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 if
  – (i) there is a path from s to t without indices i, in which case A[s][i] 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].
Transitive closure:

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

How to further improve this?

Test for A[s][i] early

Program 19.3 Warshall’s algorithm

The structure for class Graphcl computes the transitive closure of a given graph G. The class uses a two-dimensional array to represent the graph’s adjacency matrix. The algorithm runs in O(V^3) time, which is optimal. Program 19.3 outlines the implementation of the algorithm.

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

Table 19.3 Empirical study of transitive-closure algorithms

This table shows running times of two transitive-closure algorithms on random digraphs, both dense and sparse. For all but the adjacency-lists 100% representation, we used 1000 vertices in all tests, with V = 1000. The time is measured in microseconds, except for the adjacency-lists 100% representation, which is measured in milliseconds.

Random walks...

Graph:

- A > B 0.95
- A > C 0.05
- B > D 0.8
- B > E 0.3
- C > E 0.8
- 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 1.0 0
4. 1.0 0 0 0 0
5. 0.8 0 0 0.2 0

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 (1-d) times
Sum of probabilities on all pages linking to pi (d times)

Historic snapshot

Historic snapshot

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

IntAct: Protein interactions (PPI), 18773 interactions
IntAct: PPI via orthologs from IntAct, 6705 interactions
MEM: gene expression similarity over 89 tumor datasets, 46286 interactions
Transfac: gene regulation data, 5183 interactions

Module evaluation

GO: Brain development
Pigment granule
Melanine metabolic process
JAK-STAT cascade, Kinase inhibitor activity
Insulin receptor signaling pathway
KEGG: Type II diabetes mellitus
GO: Transforming growth factor beta signaling pathway, embryonic development, gastrulation
KEGG: Cell cycle, cancers, WNT pathway

MCL clustering algorithm

Stijn van Dongen

- Markov (Chain Monte Carlo) Clustering
  - 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
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

2017: Socially sensitive search

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

Landmark-based estimation

Basic Method

3 ≤ d ≤ 5

1 ≤ d ≤ 7

Landmark-based estimation

Shortest path tree
Given two nodes $U$ and $V$:
1. Collect all paths from $U$ and $V$ to all landmarks
2. Run a BFS* or Dijkstra, or A*, or anything else
Landmark-based approximation

- Basic Method
- LCA
- Shortcutting
- Landmarks-BFS
  
  Speed

Landmark-based approximation

- Basic Method
- LCA
- Shortcutting
- Landmarks-BFS
  
  Dynamic

Results

Insertion of an edge
Deletion – more complicated

Evaluation - Data

| Dataset | | | |
|---------|---|---|---|---|
| V | E | d | S/|V| t_{avg} |
| 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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Basic</td>
<td>LCA/SC/LBFS</td>
</tr>
<tr>
<td>DBLP</td>
<td>770K</td>
<td>3.0M</td>
</tr>
<tr>
<td>Orkut</td>
<td>3.0M</td>
<td>12.0M</td>
</tr>
<tr>
<td>Twitter</td>
<td>40M</td>
<td>160M</td>
</tr>
<tr>
<td>Skype</td>
<td>433M</td>
<td>1.7G</td>
</tr>
</tbody>
</table>

Timings : Query

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Method</th>
<th>No. of Landmarks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20</td>
<td>60</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>5.16</td>
</tr>
</tbody>
</table>

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

Linux, mmap, 32 cores, 256GB RAM

Timings : Updates

| | DBLP | Orkut | Twitter | Skype |
| | Insertion | Deletion* | | |
| | | | | |
| DBLP | 1μs | 100μs | 10μs | 30μs |
| Orkut | 10μs | 2ms | 12μs |
| Twitter | 10μs |
| Skype | 11μs |

* 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 B
C D
E F

Best Coverage

A B
C D
E F

Best Coverage

A B
C D
E F

Results

d' - d

Error:

Approximation Error

Number of Landmarks

<table>
<thead>
<tr>
<th>Basic</th>
<th>LCA</th>
<th>LCASC</th>
<th>UFB</th>
<th>UFS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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>34 h</td>
</tr>
</tbody>
</table>

Summary (Skype graph)

- **Network size**: 500M nodes, 3B edges
- **Landmark selection time (HD)**: 1 min / 54 hr
- **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%

Summary

- **LCA**  
- **Shortcutting**  
- **Landmarks-BFS**  

**Dynamic updates**

- **Highest degree**  
- **Best coverage**

Questions

- **LCA**  
- **Shortcutting**  
- **Landmarks-BFS**  

**Dynamic updates**

- **Highest degree**  
- **Best coverage**

Timings: Query / Twitter

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Method</th>
<th>No. of Landmarks</th>
<th>20</th>
<th>60</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Twitter</td>
<td>Basic</td>
<td></td>
<td>0.15</td>
<td>0.49</td>
<td>0.84</td>
</tr>
<tr>
<td></td>
<td>LCA</td>
<td></td>
<td>0.72</td>
<td>0.84</td>
<td>1.21</td>
</tr>
<tr>
<td></td>
<td>SC</td>
<td></td>
<td>0.83</td>
<td>0.92</td>
<td>1.87</td>
</tr>
<tr>
<td></td>
<td>LBFS</td>
<td></td>
<td>240</td>
<td>633</td>
<td>889</td>
</tr>
</tbody>
</table>

Generalizations

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

- **To directed graph:**
  - Use two SPTs per landmark
Improvements

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

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)

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)

A community is a group of densely interconnected nodes

Community Structure of Zachary Karate club using Girvan Newman

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

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.

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

Belgian mobile phone network

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


References

• http://macroconnections.media.mit.edu/wp-content/uploads/2013/01/NetworksClass5_2013.ppt

MAXIMUM FLOW

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

What is Network Flow?

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

Two distinguished vertices exist in $G$ namely:

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

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

What is Network Flow?

• Each edge $(u,v)$ has a non-negative capacity $c(u,v)$.
• If $(u,v)$ is not in $E$ assume $c(u,v)=0$.
• We have source $s$ and sink $t$.
• Assume that every vertex $v$ in $V$ is on some path from $s$ to $t$.

Following is an illustration of a network flow:
Conditions for Network Flow

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

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

Skew symmetry condition implies that \(f(u, u) = 0\).

The Value of a Flow.

The value of a flow is given by:

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

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

Example of a flow

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

<table>
<thead>
<tr>
<th></th>
<th>Flow 1</th>
<th>Flow 2</th>
<th>Flow 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(s)</td>
<td>9</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>(3)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(4)</td>
<td>6</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>(6)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(7)</td>
<td>8</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>(t)</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>(4)</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>(3)</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>(2)</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

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

Cuts of Flow Networks

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

<table>
<thead>
<tr>
<th></th>
<th>Capacity 1</th>
<th>Capacity 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(s)</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>(3)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(4)</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>(6)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(7)</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>(t)</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>(4)</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>(3)</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>(2)</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Cut

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.

Capacity = 28

Flow of Min Cut (Weak Duality)

Let f be the flow and let (S,T) be a cut. Then |f| ≤ \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:

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

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

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

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

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);
\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, v_1, v_2, ..., v_k 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.

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

```
FORDFULKERSON(G,E,s,t)
FOREACH e ∈ E
    f(e) ← 0
G_t ← residual graph
WHILE (there exists augmenting path P)
    f ← augment(f, P)
    update G_t
ENDWHILE
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 \((s,i)\) and doesn’t increase the residual capacity for some edge \((s,j)\) 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 \( G_f \) will have no more augmented paths.
2. Let \( v=\text{Max}(S,T) \) 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

![Original network](image1)

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

Find any s-t path in \( G(x) \)

![Path](image2)
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
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

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 $s$.

Here is the optimal flow
Counterexample for termination

Ellosanu 54. Værene vogej jëghesséi jënsid.

Olgy $R = x^{2}y^{2}$ (ics $R = R^{2} + R^{3}$) ja ilejand serødel enured líbhussesid. Olgy extreme serødel abel $s = a - d - 1$ sing jëghesséi serødel abel (table): 1. $s - c - f - d = a - b - c - 1$
2. $s - b - c - f - e - d - 1$
3. $s - a - d - c - b - e - f - 1$.

Nillu, et Ford-Fulkerson algoritme et Ripeta GGD.

Distribution & Transportation

Assigning teachers to classes

Teacher likes to teach C1, C4, C6

Every course will need a nr of teachers

Every teacher has a maximal capacity to teach

"Likes" – by weight

How would you solve it?

Converting the Matching problem to Network Flow

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

Converting Matching to Network Flow
Converting Optimal Bipartite Matching to Network Flow