Introducing...

- 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

Graphs

Graphs are one of the unifying themes of computer science. A graph $G = (V, E)$ is defined by a set of vertices $V$, and a set of edges $E$ consisting of ordered or unordered pairs of vertices from $V$.

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 relationships among organisms based on similarity of the primary sequences of their CYTOCHROMES c proteins

Find a shortest path from station A to station B. 
-need serious thinking to get a correct algorithm.

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 $\Delta_{\text{mut}}$

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

Filterable network $\Delta_{\text{mut}}$. Red to purple underlays in green are groups of genes which are more interconnected. The green and blue colors underlays show single and double paralogous genes. The red underlay shows genes with no known function (unknown), light blue, mitochondrial function (mitochondrial), dark blue, cleavable (cleav, purple) and a group of genes of unknown function (unkonw, gray).

Circuits
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

Dariusz Kowalski

Lines, cycles, trees, cliques

Dariusz Kowalski
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

**Hamilton Circuit**

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 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?
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.

Visualization and human computation “brain exercise”

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”?

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.

---

**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 $G$.

---

**Complete graph**

- Every node is connected to every other node
- Clique – fully connected subgraph of a graph

---

**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 $V'$
- Nr of different possible graphs of size $m,n$ is huge

---

**How many different subgraphs does a complete graph have?**

- How many?
  - $2^m$ different subsets of vertices (= many!)
  - Likewise, nr of edges is any subset of set of edges...
- 5 nodes => \( 5^4/2 \) different possible undirected edges without self-loops
  - Calculate the possibility of each edge being present or not...
  - Directed: $\rightarrow$, $\leftarrow$, $\leftrightarrow$, none (4 options)
**Representation of Graphs**

- **Adjacency list:** $\Theta(V+E)$
  - Preferred for **sparse** graph
  - $|E| \ll (V)^2$
  - Adjacency list 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:** $\Theta(V^2)$
  - 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**

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

**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.

**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.
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) // discover neighbours
    if v not yet discovered
      then 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

Example (BFS)
Example (BFS)

Q: w r

1 1

Example (BFS)

Q: r t x

1 2 2

Example (BFS)

Q: t x v

2 2 2

Example (BFS)

Q: v u y

2 3 3

Example (BFS)

Q: u v y

3 3
Example (BFS)

Example (BFS)

Example (BFS) – BF Tree

BF Tree

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 $\Rightarrow$ 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 $v$

```
PRINT-PATH(G, s, v)
1 if v = s
2 then print s
3 else if \pi(v) = NIL
4 then print "no path from" s "to" v "exists"
5 else PRINT-PATH(G, s, \pi(v))
6 print v
```
PRINT-PATH Illustration

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:

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.

process-adjacent(x, y):
  if color(y) == color(x):
    bipartite = FALSE;
    print “Warning: graph is not bipartite! (x,y);”;
  color(y) = complement(color(x));

We can assign the first vertex in any connected component to be whatever color/sex we wish.
**Problem of the Day**

Prove that in a breadth-first search on a undirected graph $G$, every edge in $G$ is either a tree edge or a cross edge, where a cross edge $(x, y)$ is an edge where $x$ is neither an ancestor or descendant of $y$.

---

**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
- DFS will create a forest of DFS-trees

---

**Simple DFS from $n$**

```plaintext
push ( Q, n )

while u = pop (Q)

process u

for each v in reverse Adjacency( u )

push ( Q, v )
```

**Recursive:**

```plaintext
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: $1 \leq d[u] < f[u] \leq 2|V|$
  - $d[u]$ 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-Visit

1. $\text{color}[u] \leftarrow \text{GRAY}$ → White vertex $u$ has just been discovered.
2. $\text{time} \leftarrow \text{time} + 1$
3. $\pi[u] \leftarrow \text{NIL}$
4. for each $v \in \text{Adj}(u)$ → Explore edge $(u, v)$.
5. do if $\text{color}[v] = \text{WHITE}$
6. then $\pi[v] \leftarrow u$
7. $\text{DFS-Visit}(v)$
8. $\text{color}[v] \leftarrow \text{BLACK}$ → Blacken $v$; 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)
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
    - Total time: \(\sum_{v \in V} |Adj(v)| = \Theta(V)\)
- 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.

```
if (discussed[v] == FALSE) {
  stack.push(v); // temporary pointer
  stack.push(v); // search marker
  if (visited) return (* allow for search termination *)
  discussed[v] = TRUE;
  discovery[v] = nsys_time();
  processed[v] = NULL;
  } else {if (visited) return;
  }
  } // end else
process(v, adj(v));

if (discussed[v] == TRUE) {
  processed[v] = TRUE;
}
```

Another Example of DFS

DFS

- Stack
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 \( u \) is a descendant of \( v \) in a depth-first tree.
3. \( d[v] < d[u] < f[u] < f[v] \) and \( v \) is a descendant of \( u \) in a depth-first tree.

- Like parentheses:
  - OK \((())[]())\)
  - Not OK \(([])[]())\)

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

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.
**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.

---

**Example (Parenthesis Theorem)**

**Depth-First Trees**

- Predecessor subgraph defined slightly different from that of BFS.
- The predecessor subgraph of DFS is $G_π = (V, E_π)$ where $E_π = \{(π[v], v) : v ∈ V \text{ and } π[v] = \text{NIL}\}$.
  
  - How does it differ from that of BFS?
  - The predecessor subgraph $G_π$ forms a **depth-first forest** composed of several **depth-first trees**. The edges in $E_π$ are called **tree edges**.

**Definition:**

Forest: An acyclic graph $G$ that may be disconnected.

---

**White-path Theorem**

Theorem 22.9

$v$ is a descendant of $u$ 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 $d[u] > d[v]$

**Theorem:**

In DFS of an undirected graph, we get only tree and back edges. No forward or cross edges.
Types of edges in DFS

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 to an ancestor
3. A forward edge to a descendant
4. A cross edge to a different set

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.

No Cross Edges in DFS

Suppose we have a cross-edge

DFS Application: Finding Cycles

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

DFS-visit – visit all reachable nodes

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

- Tree edge
- Back edge
- Forward edge
- Cross 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 df[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 $O(n(m+n))$—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 if v is not a leaf and some subtree of v has no back edge incident until a proper ancestor of v.

The root is a special case since it has no ancestors.

X is an articulation vertex since the right subtree does not have a back edge to a proper ancestor.

TOPOLOGICAL SORTING
Topological Sorting

• A directed, acyclic graph has no directed cycles.

A topological sort of a graph is an ordering on the vertices so that all edges go from left to right.

DAGs (and only DAGs) have at least one topological sort (here G, A, B, C, F, E, D).

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.

• DAGs are used in many applications to indicate precedence among events.

Topological Sort

\[ \Theta(V + E) \]

\textsc{Topological-Sort}(G)
1. call \textsc{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:

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

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

- $G^{\text{SCC}} = (V^{\text{SCC}}, E^{\text{SCC}})$.
- $V^{\text{SCC}}$ has one vertex for each SCC in $G$.
- $E^{\text{SCC}}$ has an edge if there’s an edge between the corresponding SCC’s in $G$.
- $G^{\text{SCC}}$ for the example considered:

$G^{\text{SCC}}$ is a DAG

**Lemma 22.13**

Let $C$ and $C'$ be distinct SCC's in $G$, let $u, v \in C, u', v' \in C'$, and suppose there is a path $u \sim u'$ in $G$. Then there cannot also be a path $v \sim v'$ in $G$.

**Proof:**

- Suppose there is a path $v \sim v'$ in $G$.
- Then there are paths $u \sim u' \sim v'$ and $v' \sim v \sim u$ in $G$.
- Therefore, $u$ and $v'$ are reachable from each other, so they are not in separate SCC’s.
Transposition 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

- SCC(\( G \))
  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 - 1999 (Cheriyan, Melhorn 1996)

Example

(Courtesy of Prof. Jim Anderson)

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 \( G^T \), 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)
Let $w \in E$ such that $u \in C$ and $v \in C$. Then $f(C) > f(C')$.

**Proof:**

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

- 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 $C$ are white.
  - By earlier lemma, since there is an edge $(u, v)$, we cannot have a path from $C$ to $C'$.
  - Therefore, time $d(y)$, all vertices in $C$ are white.
  - Therefore, for all $u \in C$, $f(u) = f(y)$, which implies that $f(C') > f(C)$.

The second DFS starts from some $C'$, which is a strongly connected component (SCC) of $G$. Suppose there is an edge $(u, v) \in E'$, where $u \in C$ 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 $(u, v) \in E'$, such that $u \in C$ and $v \in C$. Then $f(C) < f(C')$.

**Proof:**

- $(u, v) \in E' \Rightarrow (u, v) \in E$.
- 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**

(a)  

(b)  

(c)  

(d)
Why does strongly connected component method work?

• See CLRS (2-3 pages)


Advanced Algorithmics (6EAP)
Graphs II

Jaak Vilo
2011 Spring

WEIGHTED GRAPH ALGORITHMS

Minimum Spanning Tree

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

Fully connected graph. Find a MST?

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.

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**

**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

---

**Illustration of Theorem 23.1**

- \( A=\{(a, b), (c, i), (h, g), (g, f)\} \)
- \( S=\{a, b, c, i, s\} \); \( V-S=\{h, g, f, d\} \)
  - 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) = W(T)
• (u, v) is actually a safe edge for A
  - Since A \subseteq T and (x, y) \notin A \implies 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, fringe (meaning there exists an edge from a tree vertex to \( v \)) or unvisited (meaning \( v \) is more than one edge away).

Prim-MST(\( G \))
- Select an arbitrary vertex \( v \) to start the tree from.
- While (there are still non-tree vertices)
  - Select the edge of minimum weight between a tree and a non-tree vertex.
  - Add the selected edge and vertex to the tree \( T_{\text{rest}} \).
This creates a spanning tree, since no cycle can be introduced, but is it minimum?

Prim’s Algorithm (Cont.)

- How to efficiently select the safe edge to be added to the tree?
  - Use a min-priority queue \( Q \) that stores all vertices not in the tree
    - Based on \( \text{key}[v] \), the minimum weight of any edge connecting \( v \) to a vertex in the tree
    - \( \text{key}[v] = \infty \) if no such edge
- \( \pi[v] \) = parent of \( v \) in the tree
- \( A = \{(v, \pi[v]) : v \in V - \{r\} - Q\} \) finally \( Q = \emptyset \)

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

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

Lines 1-5 initialize the min-heap \( \text{MH} \) 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

Add the closest vertex and edge to current \( T \)
Get all adjacent vertices \( v \) of \( u \)
update \( D \) of each non-tree vertex adjacent to \( u \)
Store the current minimum weight edge and updated distance in the \( MH \)

24.10.13

35
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: $O(lg V)$
    
    - 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

```
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 \(O(m \log n)\) better than \(O(m \log m)\)?

---

**Kruskal algorithm for MST**

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

---

**Why is Kruskal’s algorithm correct?**

Again, we use proof by contradiction. Suppose Kruskal’s algorithm does not always give the minimum cost spanning tree on some graph. If so, there is a graph on which it fails. And if so, there must be a first edge \((x, y)\) Kruskal adds such that the set of edges cannot be extended into a minimum spanning tree.

When we added \((x, y)\) there previously was no path between \(x\) and \(y\), or it would have created a cycle.

Thus if we add \((x, y)\) to the optimal tree it must create a cycle.

At least one edge in this cycle must have been added after \((x, y)\), so it must have a heavier weight.

Deleting this heavy edge leave a better MST than the optimal tree? A contradiction!

---

**How fast is Kruskal’s algorithm?**

What is the simplest implementation?

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

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

---

**Fast Component Tests Give Fast MST**

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

- **Same component \((v_1, v_2)\)** – Do vertices \(v_1\) and \(v_2\) lie in the same connected component of the current graph?
- **Merge components \((C_1, C_2)\)** – Merge the given pair of connected components into one component.
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...

Problem of the Day

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

### Table 26.1 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 25.0.0). 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><code>O(V^2)</code></td>
<td>optimal for dense graphs</td>
</tr>
<tr>
<td>Prim (FPL, heap)</td>
<td><code>O(E log V)</code></td>
<td>conservative upper bound</td>
</tr>
<tr>
<td>Kruskal</td>
<td><code>O(E log E)</code></td>
<td>linear unless extremely sparse</td>
</tr>
<tr>
<td>Kruskal (partial sort)</td>
<td><code>O(E log V)</code></td>
<td>cost depends on longest edge</td>
</tr>
<tr>
<td>Boruvka</td>
<td><code>O(V)</code></td>
<td>conservative upper bound</td>
</tr>
</tbody>
</table>

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 $w : E \rightarrow \mathbb{R}$, the weight of a path $p=v_0v_1...v_k$ is the sum of the weights of its constituent edges: $w(p) = \sum_{i=0}^{k} w((v_i, v_{i+1}))$

- We define the shortest-path weight from $u$ to $v$ by $\delta(u, v) = \min\{w(p) : p \text{ is a path from } u \text{ to } v\}$, if there is a path from $u$ to $v$.

- Otherwise, $\delta(u, v) = \infty$.

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

Variants

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

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

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

- **All-pairs shortest-path problem** — dynamic programming
  - Can be solved faster than running the single-source shortest-path problem for each vertex

Optimal Substructure of A Shortest-Path

- **Lemma 24.1** (Subpath of shortest paths are shortest paths). Let $p=v_0v_1...v_k$ be a shortest path from vertex $v_i$ to $v_j$, and for any $i$ and $j$ such that $1 \leq i \leq j \leq k$, let $p_{ij} = v_{i+1}v_{i+2}...v_j$ be the subpath of $p$ from vertex $v_i$ to $v_j$.

  Then $p_{ij}$ is a shortest path from vertex $v_i$ to $v_j$. 

---

Weighting the Graph

The weight of each edge is a function of the probability that these two words will be next to each other in a sentence. ‘hive me’ would be less than ‘give me’, for example. The final system worked extremely well — identifying over 99% of characters correctly based on grammatical and statistical constraints.
Examples of shortest paths depending on start node

Negative-Weight Edges and Cycles
- Cannot contain a negative-weight cycle
- Of course, a shortest path cannot contain a positive-weight cycle

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

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

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 \( \pi[v] \).

**Relax** \((u, v, w)\)
1. if \( v.d > u.d + w(u,v) \)
2. \( v.d \leftarrow u.d + w(u,v) \)
3. \( \pi[v] \leftarrow u \)

Bellman-Ford

Bellman-Ford \((G, w, s)\)
1. **INITIALIZE-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

**RELAX** \((u, v, w)\)
1. if \( v.d > u.d + w(u,v) \)
2. \( v.d \leftarrow u.d + w(u,v) \)
3. \( \pi[v] \leftarrow u \)
Bellman-Ford

• \(O(V E)\)

• Just repeatedly relax all edges.
  – Allow \(V\) cycles to propagate through the network

Shortest paths on a DAG

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

\(O(V + E)\)

Dijkstra’s Algorithm

• Solve the single-source shortest-paths problem on a weighted, directed graph 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 \(s\) 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. \(\text{while } Q \neq \emptyset\)
5. \(\text{do } u \leftarrow \text{EXTRACT-MIN}(Q)\)
6. \(S \leftarrow S \cup \{u\}\)
7. \(\text{for each vertex } v \in Adj[u]\)
8. \(\text{do } \text{RELAX}(u, v, w)\)

Note: \(\text{relax} \) requires updating of min values in \(Q\)
Analysis of Dijkstra’s Algorithm

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

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

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

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

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

ACM Conference on Information and Knowledge Management (CIKM) 2011

Fast Fully Dynamic Landmark-based Estimation of Shortest Path Distances in Very Large Graphs
Fast **Fully Dynamic** Landmark-based Estimation of Shortest Path Distances in Very Large Graphs

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

Basic Method

1 <= d <= 7
3 <= d <= 5

Least common ancestor

Least common ancestor

Shortcutting
Combining multiple landmarks

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

Dynamic
Basic Method
LCA
Shortcutting
Landmarks-BFS
Insertion of an edge
Deletion – more complicated

Evaluation - Data

<table>
<thead>
<tr>
<th>Dataset</th>
<th></th>
<th></th>
<th></th>
<th>t_{RFS}</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLP</td>
<td>774K</td>
<td>2.6M</td>
<td>6.3</td>
<td>23</td>
</tr>
<tr>
<td>Orkut</td>
<td>3.1M</td>
<td>11.7M</td>
<td>5.7</td>
<td>10</td>
</tr>
<tr>
<td>Twitter</td>
<td>41.7M</td>
<td>1.2B</td>
<td>4.2</td>
<td>24</td>
</tr>
<tr>
<td>Skype</td>
<td>454M</td>
<td>3.1B</td>
<td>6.5</td>
<td>50</td>
</tr>
</tbody>
</table>

Timings:

**Queries**

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Graph file</th>
<th>Landmark file</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLP</td>
<td>27M</td>
<td>Basic LCA/LBFS</td>
</tr>
<tr>
<td>Orkut</td>
<td>908M</td>
<td>12.0M</td>
</tr>
<tr>
<td>Twitter</td>
<td>93G</td>
<td>100M</td>
</tr>
<tr>
<td>Skype</td>
<td>27G</td>
<td>1.7G</td>
</tr>
</tbody>
</table>

**Updates**

- Insertion: 1μs
- Deletion*: 100μs

* very non-uniform

Timings: Query

<table>
<thead>
<tr>
<th>Dataset</th>
<th>No. of Landmarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skype</td>
<td>Basic</td>
</tr>
<tr>
<td></td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>0.18</td>
</tr>
</tbody>
</table>

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

Linux, mmap, 32 cores, 256GB RAM

Outline

- **Improvement to Basic Landmark method**
- **Dynamic updates**

**Landmark selection method**

- Landmark is good if it covers many shortest paths

**Evaluation**

- Highest degree
- Best coverage
24.10.13

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>12 s</td>
<td>15 min</td>
</tr>
<tr>
<td>Skype</td>
<td>1 min</td>
<td>54 h</td>
</tr>
</tbody>
</table>

Summary (Skype graph)

- Network size: 500M nodes 3B edges
- Landmark selection time (HD): 1 min / 54 hr
- Landmark computation time: 20 min x 100
- Total space for 100 landmarks: 170G
- Avg query time (SC/LBFS): 5 ms / 16 ms
- 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

- Precomp: $1 + 100 \times 20 \text{min}$
- Space: 170G
- Query: 5ms / 16ms
- Insertion: 0.030 ms
- Deletion: 11ms
- Error: 18% / 15%

Generalizations

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

- To directed graph:
  - Use two SPTs per landmark

Improvements

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

Timings: Query / Twitter

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Method</th>
<th>No. of Landmarks</th>
</tr>
</thead>
<tbody>
<tr>
<td></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>240</td>
</tr>
</tbody>
</table>

Euclidean 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

\[ c_{ij} = \sum_k a_{ik} b_{kj} \]
\[ c_{st} = \sum_i a_{si} b_{it} \]

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

**Diagonal 1 = self-loop**

**Transitive closure**

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

**Diagonal 0 or 1**

**G*G**

The textbook algorithm for computing the product of two \(V\times V\) matrices computes, for each \(s\) and \(t\), the dot product of row \(s\) and row \(t\) in the first matrix and row \(s\) in the second matrix, as follows:

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

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 \(*, +, \text{ and } \,*\) are defined. In particular, if the matrix entries are either \(true\) or \(false\) and we interpret \(*\) to be the logical \(\&\) operator and \(+\) to be the logical \(\lor\) operator, 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 \((A[s][i] \&\& B[i][t]) \) \(C[s][t] = true;\)
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^4*V^4=V^8, ... => O( [log V] * V^3 )
- Can we avoid so many cycles?

Multiply:

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

Transitive closure:

```plaintext
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;
```

Book: Sedgewick, Algorithms ...

- 19.3. Reachability and Transitive Closure

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^v.

Proof: Considering the argument in the previous paragraph, A^v has an edge for every path of length less than or equal to v in the digraph.
A^v has no edge for every path of length less than or equal to v 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 contain some vertex that occurs on only v of them and therefore adds no new information to the transitive closure because the same two vertices are connected by a direct path of length less than v (which we could obtain by removing the cycle from the oriented vertex).

Paths via 0
Paths via 1 (including 0-1, 1-0)
• Proof: transitive closure by induction on \(i\).
  
  Iteration 1: either \(s\)-t or the path \(s\-1\-t\).
  
  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 \(s\-0\-t\).
  
  Inductive hypothesis: the \(i\)th iteration of the loop sets the bit \((s, t)\) to true if there is a directed path from \(s\) to \(t\) in the digraph that does not include any vertices with indices greater than \(i\) (except possibly the endpoints \(s\) and \(t\)).

- How to further improve?
- Test for \(A[s][i]\) early

Example of Wartels Algorithm

```java
class DigraphV0
{
    private String V;
    private Digraph<T> G;
    private int[][] A;

    // Constructor
    DigraphV0(String V, Digraph<T> G, int[][] A)
    {
        this.V = V;
        this.G = G;
        this.A = A;
    }

    // Method to check if a directed path exists from s to t
    boolean reachable(int s, int t)
    {
        return A[s][t] == true;
    }
}
```
Random walks...

Graph:
- $A \rightarrow B$ 0.95
- $A \rightarrow C$ 0.05
- $B \rightarrow D$ 0.7
- $B \rightarrow E$ 0.3
- $C \rightarrow B$ 1.0
- $D \rightarrow A$ 1.0
- $E \rightarrow D$ 0.2
- $E \rightarrow A$ 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

Finding the modules

IntAct: Protein interactions (PPI), 18,773 interactions
IntAct: PPI via orthologs from IntAct, 6,705 interactions
MEM: gene expression similarity over 89 tumor datasets, 46,286 interactions
Transfac: gene regulation data, 5,183 interactions

Module evaluation

Public datasets for H. sapiens
- IntAct: Protein interactions (PPI), 18,773 interactions
- IntAct: PPI via orthologs from IntAct, 6,705 interactions
- MEM: gene expression similarity over 89 tumor datasets, 46,286 interactions
- Transfac: gene regulation data, 5,183 interactions
MCL clustering algorithm

- Markov (Chain Monte Carlo) Clustering
  - [http://www.micans.org/mcl/](http://www.micans.org/mcl/)
- Random walks according to edge weights
- Follow the different paths according to their probability
- Regions that are traversed “often” form clusters

http://www.micans.org/mcl/intro.html

With this, the MCL algorithm can be written as:

```c
G is a graph
add loops to G
set r to some value
set H[1] to be the matrix of random walks on G
while change (

  \[ H_{i,j} = H_{i,j} + H_{i,j} \times H_{i,j} \]
  \[ \text{inflation} \]

  change = difference(H_{i,j},H_{i,j})
)
set CLUSTERING as the components of H[1]
```

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)≥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≤f(u,v)+f(v,u)≤c(u,v), for every edge (u,v)∈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:

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

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>Flow</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

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

The Maximum Flow Problem

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

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

Maximize \(\sum_{i} x_{s,i}\)
Subject To
- \(\sum_{j} x_{i,j} - \sum_{j} x_{j,i} = 0\) for each \(i \neq s,t\)
- \(\sum_{i:j \in \text{in}(s)} x_{s,i} = v\)
- \(0 \leq x_{ij} \leq u_{ij}\) for all \((i,j) \in E\).

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

Cuts of Flow Networks

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

Table illustrating Cuts and Capacity across different cuts of graph above:

<table>
<thead>
<tr>
<th>Cut</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-2</td>
<td>10</td>
</tr>
<tr>
<td>3-4</td>
<td>30</td>
</tr>
</tbody>
</table>

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

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

In the graph above, the capacity of the cut is 30.
Min Cut

Min s-t Cut (Also called as a Min Cut) is a cut of minimum capacity

Flow of Min Cut (Weak Duality)

Let f be the flow and let (S, T) be a cut. Then \( |f| \leq \text{CAP}(S,T) \).

In maximum flow, minimum cut problems forward edges are full or saturated and the backward edges are empty because of the maximum flow. Thus maximum flow is equal to capacity of cut. This is referred to as weak duality.

Proof:

\[
|f| = \sum_{e \in E} f(e) = \sum_{e \in S} f(e) + \sum_{e \in T} f(e) \\
= \sum_{e \in S} c(e) = \text{cap}(S, T)
\]

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:

\[
c_p(e) = c(e) - f(e) \quad \text{if} (i,j) \text{ is a forward edge, and}
\]

\[
c_p(e) = f(e) \quad \text{if} (i,j) \text{ is a backward edge.}
\]

Moreover the residual capacity of an edge is always non-negative.

Augmenting Paths (A Useful Concept)

Definition:

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

Characteristics of augmenting paths:

- We can put more flow from \( s \) to \( t \) through \( p \).
- The edges of 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 \( p \).

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

```
FORDFULKERSON(G,E,s,t)
FOREACH e ∈ E
    f(e) ← 0
G_f ← residual graph
WHILE (there exists augmenting path P)
    f ← augment(f, P)
    update G_f
ENDWHILE
RETURN f
```

AUGMENT(f,P)

```
b ← bottleneck(P)
FOREACH e ∈ P
    IF (e ∈ E)
        // backwards arc
        f(e) ← f(e) + b
    ELSE
        // forward arc
        f(e^R) ← f(e) - b
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 Δ of P is the smallest residual capacity on P which is integral. After updating, we modify the residual capacities by 0 or Δ, and thus residual capacities stay integers.

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

**Proof:** The capacity of each augmenting path is at least 1. The augmentation reduces the residual capacity of some edge (s,j) and doesn't increase the residual capacity for some edge (s,i) for any i. So the sum of residual capacities of edges out of s keeps decreasing, and is bounded below 0. Number of augmentations is O(nC) where C is the largest of the capacity in the network.

When is the flow optimal?

A flow f is maximum in G if:

1. The residual network G_f contains no more augmented paths.
2. |f| = 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 network G_f will have no more augmented paths.

2. Let v=Fx(S,T) be the flow from s to t. By assumption v=CAP(S,T) By Weak duality, the maximum flow is at most 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

```
3 2
1
5
3 4
2
1

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

Find any s-t path in G(x)

```
3 2
1
2
3 4
5
3

Ford-Fulkerson Max Flow
```

1

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.

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

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

Ford-Fulkerson Max Flow

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

Counterexample 54. We define a graph for the problem of finding a maximum matching in a bipartite graph. The graph consists of two sets of vertices, one for each side of the bipartite graph, and edges connecting vertices from one set to the other.

Counterexample

Distribution & Transportation

Assigning teachers to classes

Assigning teachers to classes involves assigning teachers to classes based on their preferences and the number of teachers required for each course.

1. Every teacher likes C1, C4, C6.
2. Every course will need a certain number of teachers.
3. Every teacher has a maximal capacity to teach.

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

Job placement involves matching people with suitable job positions based on their preferences and the requirements of the positions.

Converting the Matching problem to Network Flow

Converting the Matching problem to Network Flow involves transforming a matching problem into a network flow problem, which can then be solved using algorithms such as the Ford-Fulkerson algorithm.

Converting Matching to Network Flow

Converting Matching to Network Flow involves transforming a matching problem into a network flow problem, which can then be solved using algorithms such as the Ford-Fulkerson algorithm.
Converting Optimal Bipartite Matching to Network Flow