Chapter 22
Elementary Graph Algorithms

http://www.cc.nctu.edu.tw/~claven/course/Algorithm/
Lecture 10:
Graph Data Structures

Steven Skiena

Department of Computer Science
State University of New York
Stony Brook, NY 11794–4400

http://www.cs.sunysb.edu/~skiena

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 relationship among organisms based on similarity of the primary sequences of their CYTOCHROME c proteins

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

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

Mutation network $\Delta_\gamma = 4$
Probability network \( \Delta \gamma = 2 \), underlayed in green are groups of genes which are more interconnected. The genes are coloured according to annotation in YPD ('cellular role'). The genes which are more interconnected are involved in the same cellular processes, like mating behaviour (mat, green), aminoacid metabolism (aam, red), cos gene family (cos, light blue), mitochondrial function (mitochondrial, dark blue), ribosome (ribo, purple) and a group of genes of unknown function (unknown, grey).
Find a shortest path from station A to station B. - need serious thinking to get a correct algorithm.

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

http://www.cc.nctu.edu.tw/~claven/course/Algorithm/
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
- 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

Lines, cycles, trees, cliques

- Line
- Cycle
- Clique
- Tree
A joke:

• 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

(a) Konigsberg graph

(b)
A joke (continued):

- **Why?** no body in the company has taken CS4335.
- **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.
"I can’t find an efficient algorithm, I guess I’m just too dumb."

"I can’t find an efficient algorithm, because no such algorithm is possible!"
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.
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.

---

**Figure 17.3**

Graph terminology

This graph has 55 vertices, 70 edges, and 3 connected components. One of the connected components is a tree (right). The graph has many cycles, one of which is highlighted in the large connected component (left). The diagram also depicts a spanning tree in the small connected component (center). The graph as a whole does not have a spanning tree, because it is not connected.
Complete graph

• Every node is connected to every other node

• Clique – fully connected subgraph of a graph

• How many different subgraphs does a complete graph have?

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 $V$

• How many?
  – $2^m$ different subsets of vertices (= many!)
  – Likewise, nr of edges is any subset of set of edges...

• Nr of different possible graphs of size $m,n$ is huge
Representation of Graphs

- **Adjacency list:** \( \Theta(V+E) \)
  - Preferred for *sparse* graph
  - \( |E| \ll |V|^2 \)
  - \( \text{Adj}[u] \) 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 \( \text{Adj}[u] \)
  - 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

---

**Representation For An Undirected Graph**

![Figure 2.2.1](image-url)

*Figure 2.2.1* Two representations of an undirected graph. (a) An undirected graph \( G \) having five vertices and seven edges. (b) An adjacency-list representation of \( G \). (c) The adjacency-matrix representation of \( G \).
Representation For A Directed Graph

(a) A directed graph $G$ having six vertices and eight edges. (b) An adjacency-list representation of $G$. (c) The adjacency-matrix representation of $G$.

Tradeoffs Between Adjacency Lists and Adjacency Matrices

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Winner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Faster to test if $(x, y)$ exists?</td>
<td>matrices</td>
</tr>
<tr>
<td>Faster to find vertex degree?</td>
<td>lists</td>
</tr>
<tr>
<td>Less memory on small graphs?</td>
<td>lists $(m+n)$ vs. $(n^2)$</td>
</tr>
<tr>
<td>Less memory on big graphs?</td>
<td>matrices (small win)</td>
</tr>
<tr>
<td>Edge insertion or deletion?</td>
<td>matrices $O(1)$</td>
</tr>
<tr>
<td>Faster to traverse the graph?</td>
<td>lists $m + n$ vs. $n^2$</td>
</tr>
<tr>
<td>Better for most problems?</td>
<td>lists</td>
</tr>
</tbody>
</table>

Both representations are very useful and have different properties, although adjacency lists are probably better for most problems.
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 \( k+1 \)

Data Structure for BFS

- Adjacency list
- \( \text{color}[u] \) for each vertex
  - \text{WHITE} if \( u \) has not been discovered
  - \text{BLACK} if \( u \) and all its adjacent vertices have been discovered
  - \text{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 \( u \)
- \( \pi[u] \) for predecessor of \( u \)
- FIFO queue \( Q \) to manage the set of gray vertices
  - \( Q \) stores all the gray vertices
BFS($G, s$)

1. for each vertex $u \in V[G] - \{s\}$
2. do $\text{color}[u] \leftarrow \text{WHITE}$
3. $d[u] \leftarrow \infty$
4. $\pi [u] \leftarrow \text{NIL}$

5. $\text{color}[s] \leftarrow \text{GRAY}$
6. $d[s] \leftarrow 0$
7. $\pi [s] \leftarrow \text{NIL}$
8. $Q \leftarrow \emptyset$
9. ENQUEUE($Q, s$)
10. while $Q \neq \emptyset$
11. do $u \leftarrow \text{DEQUEUE}(Q)$
12. for each $v \in \text{Adj}[u]$
13. do if $\text{color}[v] = \text{WHITE}$
14. then $\text{color}[v] \leftarrow \text{GRAY}$
15. $d[v] \leftarrow d[u] + 1$
16. $\pi [v] \leftarrow u$
17. ENQUEUE($Q, v$)
18. $\text{color}[u] \leftarrow \text{BLACK}$

Initialization

Set up $s$ and initialize $Q$

Explore all the vertices adjacent to $u$ and update $d, \pi$ and $Q$
Analysis of BFS

• $O(V+E)$
  – Each vertex is enqueued ($O(1)$) at most once $\Rightarrow O(V)$
  – Each adjacency list is scanned at most once $\Rightarrow O(E)$

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

```
PRINT-PATH(G, s, u)
PRINT-PATH(G, s, t)
PRINT-PATH(G, s, w)
PRINT-PATH(G, s, s)
print s
print w
print t
print u

Output: s w t u
```

### Data Structures for BFS

We use two Boolean arrays to maintain our knowledge about each vertex in the graph.

A vertex is discovered the first time we visit it.

A vertex is considered processed after we have traversed all outgoing edges from it.

Once a vertex is discovered, it is placed on a FIFO queue. Thus the oldest vertices / closest to the root are expanded first.

```c
bool processed[MAXV];
bool discovered[MAXV];
int parent[MAXV];
```
Initializing BFS

initialize_search(graph *g)
{
    int i;

    for (i=1; i<=g->num Vertices; i++) {
        processed[i] = discovered[i] = FALSE;
        parent[i] = -1;
    }
}

BFS Implementation

bfs (graph *g, int start)
{
    queue q;
    int i;
    int y;
    edgenode *p;

    init_queue(&q);
    enqueue(q, start);
    discovered[start] = TRUE;

    while (!empty_queue(&q))
    {
        v = dequeue(&q);
        processed[v] = TRUE;
        p = g->edges[v];
        while (p != NULL) {
            y = p->y;
            if ((processed[y] == FALSE) || g->directed) {
                process_edge(v, y);
                if (discovered[y] == FALSE) {

```
BFS Example

```
enqueue(&q,y);
    discovered[y] = TRUE;
parent[y] = v;
    p = p->next;
    process_vertex_list(v);
```
**Shortest Paths and BFS**

In BFS vertices are discovered in order of increasing distance from the root, so this tree has a very important property. The unique tree path from the root to any node $x \in V$ uses the smallest number of edges (or equivalently, intermediate nodes) possible on any root-to-$x$ path in the graph.

**Recursion and Path Finding**

We can reconstruct this path by following the chain of ancestors from $x$ to the root. Note that we have to work backward. We cannot find the path from the root to $x$, since that does not follow the direction of the parent pointers. Instead, we must find the path from $x$ to the root.

```c
find_path(int start, int end, int parents[])
{
    if ((start == end) || (end == -1))
        printf("%d", start);
    else
    {
        find_path(start, parents[end], parents);
        printf("%d", end);
    }
}
```
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:

Implementation

```c
connected_components(graph *g)
{
    int c;
    int i;

    initialize_search(g);
    c = 0;
    for (i=1; i<=n; i++)
    {
        if (discovered[i] == FALSE) {
            c = c+1;
            printf(“Component %d: “,c);
            bfs(g,i);
        }
    }
}
```
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.

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.

twocolor(graph *g)
{
    int i;

    for (i=1; i<=g->vertices; i++)
        color[i] = UNCOLORED;

    bipartite = TRUE;

    initialize_search(&a);

    for (i=1; i<=g->vertices; i++)
        if (discovered[i] == FALSE) {
            color[i] = WHITE;
            bfs(g,i);
        }
}
process_edge(int x, int y)
{
    if (color[x] == color[y]) {
        bipartite = FALSE;
        printf("Warning: graph not bipartite, due to (%d,%d),\n",x,y);
    }
    color[y] = complement(color[x]);
}

complement(int color)
{
    if (color == WHITE) return(BLACK);
    if (color == BLACK) return(WHITE);
    return(UNCOLORED);
}

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 descendent 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
  - The entire process is repeated until all vertices are discovered
- DFS will create a forest of DFS-trees

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.

```c
dfs(graph *g, int v)
{
    edgenode *p; (* temporary pointer *)
    int y; (* successor vertex *)

    if (finished) return; (* allow for search termination *)
    discovered[v] = TRUE;
    time = time + 1;
    entry_time[v] = time;
    process_vertex_early(v);
    p = g – > edges[v];
    while (p != NULL) {
        y = p – > y;
```
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:
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
DFS

- Stack

BFS

- Queue
Randomised search

Use:
Randomized Queue

Another Example of DFS
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.

Classification of Edges

- Tree edges are edges in the DFS forest. Edge \((u, v)\) is a tree edge if it was first discovered by exploring edge \((u, v)\).
  - \( v \) is WHITE
- Back edges are those edges \((u, v)\) connecting a vertex \( u \) to an ancestor \( v \) in a DFS tree. Self-loops, which may occur in directed graphs, are considered to be back edges.
  - \( v \) is GRAY
- Forward edges are those non-tree edges \((u, v)\) containing a vertex \( u \) to a descendant \( v \) in a DFS tree
  - \( v \) is BLACK and \( d[u] < d[v] \)
- Cross edges are all other edges. They can go between vertices in the same tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different DFS trees.
  - \( v \) is BLACK and \( d[u] > d[v] \)
- In a depth-first search of an undirected graph \( G \), every edge of \( G \) is either a tree edge or a back edge. (Theorem 22.10)
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 node

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

![Graph showing a cross-edge example]

When expanding 2, we would discover 1, so the tree would look like:

![Updated graph]

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 \( y \) creates a cycle with the path in the tree from \( y \) to \( x \).

```plaintext
process.edge(int x, int y) {
    if (parent[x] != y) { /* found back edge! */
        printf("Cycle from %d to %d via %d\n", x, y, parent[x]);
        find_path(x, parent);
        finished = TRUE;
    }
}
```
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 iff $v$ 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 ancestor. $x$ is an articulation vertex since the right subtree does not have a back edge to a proper ancestor.
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) has 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 is not acyclic, then 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

- $\Theta(V+E)$

**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
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.

Theorem Topological sort (22.12)

• 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] < [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 iff 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.

1. Use two depth-first searches, one on $G$ and one on $G^T$:
   - Transpose of $G$: $G^T = (V, E^T)$, where $E^T = \{(u, v): (v, u) \in E\}$
2. $\Theta(V+E)$
3. Component graph $G^{SCC} = (V^{SCC}, E^{SCC})$ (DAG)
   - one vertex for each component
   - $(u, v) \in E^{SCC}$ if there exists at least one directed edge from the corresponding components

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

Example

Strongly Connected Components Algorithm

STRONGLY-CONNECTED-COMPONENTS (G)
1 call DFS(G) to compute finishing times \( f[u] \) for each vertex \( u \)
2 compute \( G^T \)
3 call DFS\( (G^T) \), but in the main loop of DFS, consider the vertices in order of decreasing \( f[u] \) (as computed in line 1)
4 output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component
Why does strongly connected component method work?

• Seel CLRS (2-3 pages)


Problems 22.2: 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 $G$. A bridge of $G$ is an edge whose removal disconnects $G$. A biconnected component of $G$ is a maximal set of edges such that any two edges in the set lie on a common simple cycle. Figure 22.10 illustrates these definitions. We can determine articulation points, bridges, and biconnected components using depth-first search. Let $G_p = (V, E_p)$ be a depth-first tree of $G$.

Figure 22.10: The articulation points, bridges, and biconnected components of a connected, undirected graph for use in Problem 22-2. The articulation points are the heavily shaded vertices, the bridges are the heavily shaded edges, and the biconnected components are the edges in the shaded regions, with a bcc numbering shown.

a. Prove that the root of $G_p$ is an articulation point of $G$ if and only if it has at least two children in $G_p$.

b. Let $v$ be a nonroot vertex of $G_p$. Prove that $v$ is an articulation point of $G$ if and only if $v$ has a child $s$ such that there is no back edge from $s$ or any descendant of $s$ to a proper ancestor of $v$.

...
**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 graph (a)

(a)  

(b)  

(c)
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.

Prism’s Algorithm

If $G$ is connected, every vertex will appear in the minimum spanning tree. If not, we can talk about a minimum spanning forest. Prim’s algorithm starts from one vertex and grows the rest of the tree an edge at a time. As a greedy algorithm, which edge should we pick? The cheapest edge with which can grow the tree by one vertex without creating a cycle.
**Prim’s Algorithm (Pseudocode)**

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

**Prim-MST(G)**

Select an arbitrary vertex \( s \) 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 \( T_{prim} \).

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

---

**Prim’s Algorithm in Action**

![Prim’s Algorithm in Action](image.png)
Growing a Minimum Spanning Tree (MST)

- Generic algorithm
  - Grow MST one edge at a time
  - Manage a set of edges \( A \), maintaining the following loop invariant:
    - Prior to each iteration, \( A \) is a subset of some MST
    - At each iteration, we determine an edge \((u, v)\) that can be added to \( A \) without violating this invariant
      - \( A \cup \{u, v\} \) is also a subset of a MST
      - \((u, v)\) is called a **safe edge** for \( A \)

**GENERIC-MST**

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

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

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

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

![Figure 23.2](image)

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

- \( A = \{(a, b), (c, i), (h, g), (g, h)\} \)
- \( S = \{a, b, c, i\}; V-S = \{h, g, f, d\} \)
- 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 $A$

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

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

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

Prim’s Algorithm

• The edges in the set A always forms a single tree
• The tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V
• At each step, a light edge is added to the tree A that connects A to an isolated vertex of $G_A=(V, A)$
• Greedy since the tree is augmented at each step with an edge that contributes the minimum amount possible to the tree’s weight
Prim’s Algorithm (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
  
• $\pi[v] = \text{parent of } v \text{ in the tree}$
  
• $A = \{(v, \pi[v]): v \in V - \{r\} - Q\} \implies \text{finally } Q = \text{empty}$
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. \( D[ u ] \leftarrow \infty \)
3. \( D[ r ] \leftarrow 0 \)
4. \( MH \leftarrow \text{make-heap}(D,V,\{}\) //No edges
5. \( T \leftarrow \emptyset \)
6. \( \)  
7. while \( MH \neq \emptyset \) do  
8. \( (u,e) \leftarrow MH.\text{extractMin}() \)
9. \( \text{add} (u,e) \text{ to } T \)
10. for each \( v \in \text{Adjacent } (u) \)
11. \( \)  
12. \( \)  
13. \( \text{MH.\text{decreaseDistance}}(D[v], v, (u,v)) \)
14. return \( T \) // \( T \) is a MST

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

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

Properties of MST-PRIM

- Prior to each iteration of the while loop of lines 6—11
  - \(A = \{(v, \pi[v]): v \in V-\{r\}-Q\}\)
  - The vertices already placed into the MST are those in \(V-Q\)
  - For all vertices \(v \in Q\), if \(\pi[v] \neq \text{NIL}\), then \(key[v] < \infty\) and \(key[v]\) is the weight of a light edge \((v, \pi[v])\) connecting \(v\) to some vertex already placed into the MST
- Line 7: identify a vertex \(u \in Q\) incident on a light edge crossing \((V-Q, Q)\) \(\rightarrow\) add \(u\) to \(V-Q\) and \((u, \pi[u])\) to \(A\)
- Lines 8—11: update key and \(\pi\) of every vertex \(v\) adjacent to \(u\) but not in the tree
Illustration of MST-PRIM

- u=a, adj[a] = {b, h}
  \[\pi[b] = \pi[h] = a\]
  key[b] = 4; key[h]=8

- u=b, adj[b] = {a, c, h}
  \[\pi[h] = a; \pi[d] = b\]
  key[h]=8; key[c] = 8

- u=c, adj[c] = {b, i, d, f}
  \[\pi[d] = \pi[i] = \pi[f] = c\]
  key[h]=8; key[d] = 7; key[i] = 2; key[f] = 4

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.

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.
Fast Kruskal Implementation

Put the edges in a heap
\( count = 0 \)
while \( count < n - 1 \) do
  get next edge \((v, w)\)
  if \( \text{component}(v) \neq \text{component}(w) \)
    add to \( T \)
    \( \text{component}(v) = \text{component}(w) \)
If we can test components in \( O(\log n) \), we can find the MST in \( O(m \log m) \)!
Question: Is \( O(m \log n) \) better than \( O(m \log m) \)?

Union-Find Programs

We need a data structure for maintaining sets which can test if two elements are in the same and merge two sets together. These can be implemented by 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 \( \text{find}(i) \) now equals \( \text{find}(j) \).

See the lecture on trees…
This path compression will let us do better than $O(n \log n)$ for $n$ union-finds.

$O(n)$? Not quite... Difficult analysis shows that it takes $O(n \alpha(n))$ time, where $\alpha(n)$ is the inverse Ackerman function and $\alpha(n) \approx \log^* n = \log(n \log(n \cdots \log n))$ for $n$ at least 297.
Table 20.1 Cost of MST algorithms

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

<table>
<thead>
<tr>
<th>algorithm</th>
<th>worst-case cost</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prim (standard)</td>
<td>$V^2$</td>
<td>optimal for dense graphs</td>
</tr>
<tr>
<td>Prim (PFS, heap)</td>
<td>$E \log V$</td>
<td>conservative upper bound</td>
</tr>
<tr>
<td>Prim (PFS, d-heap)</td>
<td>$E \log_d V$</td>
<td>linear unless extremely sparse</td>
</tr>
<tr>
<td>Kruskal</td>
<td>$E \log E$</td>
<td>sort cost dominates</td>
</tr>
<tr>
<td>Kruskal (partial sort)</td>
<td>$E + X \log V$</td>
<td>cost depends on longest edge</td>
</tr>
<tr>
<td>Boruvka</td>
<td>$E \log V$</td>
<td>conservative upper bound</td>
</tr>
</tbody>
</table>
Single-Source Shortest Paths
(Chapter 24)

Shortest Paths

Finding the shortest path between two nodes in a graph arises in many different applications:

- Transportation problems – finding the cheapest way to travel between two locations.
- Motion planning – what is the most natural way for a cartoon character to move about a simulated environment.
- Communications problems – how long will it take for a message to get between two places? Which two locations are furthest apart, i.e. what is the diameter of the network.
Example: Predictive Mobile text Entry Messaging...

What was the message?

INPUT

GIVE ME A RING.
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.
Problem Definition

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

• We define the shortest-path weight from \( u \) to \( v \) by
  \[
  \delta(u, v) = \begin{cases} 
  \min \{ w(p) : u \rightarrow^p v \} & \text{if there is a path from } u \text{ to } v, \\
  \infty & \text{otherwise}.
  \end{cases}
  \]

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

Variants

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

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

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

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

- Lemma 24.1 (Subpath of shortest paths are shortest paths). Let $p=<v_1, v_2, ..., v_k>$ be a shortest path from vertex $v_1$ to $v_k$, and for any $i$ and $j$ such that $1 \leq i \leq j \leq k$, let $p_{ij} = <v_{i1}, v_2, ..., v_j>$ be the subpath of $p$ from vertex $v_i$ to $v_j$. Then $p_{ij}$ is a shortest path from vertex $v_i$ to $v_j$.

Negative-Weight Edges and Cycles

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

\[
\text{INITIALIZE-SINGLE-SOURCE}(G, s)
\]

1. for each vertex \( v \in V[G] \)
2. do \( d[v] \leftarrow \infty \)
3. \( \pi[v] \leftarrow \text{NIL} \)
4. \( d[s] \leftarrow 0 \)

Predecessor of \( v \) in the shortest path

Relaxation (Cont.)

• Relaxing an edge \((u, v)\) consists of testing whether we can improve the shortest path found so far by going through \( u \) and, if so, update \( d[v] \) and \( \pi[v] \)

![Figure 24.3](image)

By Triangle Inequality
Dijkstra’s Algorithm

• Solve the single-source shortest-paths problem on a weighted, directed graph and 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, adds $u$ to $S$, and relaxes 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. \hspace{1em} \textbf{do} $u \leftarrow \text{EXTRACT-MIN}(Q)$
6. \hspace{2em} $S \leftarrow S \cup \{u\}$
7. \hspace{2em} \textbf{for each vertex} $v \in \text{Adj}[u]$
8. \hspace{3em} \textbf{do} \text{RELAX}(u, v, w)
Analysis of Dijkstra’s Algorithm

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

- Edsger Wybe Dijkstra was one of the most influential members of computing science’s founding generation. Among the domains in which his scientific contributions are fundamental are:
  - algorithm design
  - programming languages
  - program design
  - operating systems
  - distributed processing
  - formal specification and verification
  - design of mathematical arguments

Examples of shortest paths depending on start node
All pairs shortest paths

• Diameter of a graph (longest shortest path)

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

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

Exists link via \( j \)

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

```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];
```

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

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

Complexity...

- \( V^3 \) operations for \( V^2, V^3, \ldots, V^v \)
- \( O(V^4) \)
- ceiling(log \( v \)) times \( V^3 \)
- Can we avoid so many cycles?
§19.3

CHAPTER NINETEEN

closure with just one operation of this kind, building up the transitive closure from the adjacency matrix in place, as follows:

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;

This classical method, invented by S. Warshall in 1962, is the method of choice for computing the transitive closure of dense digraphs. The code is similar to the code that we might try to use to square a Boolean matrix in place: The difference (which is significant) lies in the order of the for loops.

Property 19.7 With Warshall's algorithm, we can compute the transitive closure of a digraph in time proportional to $V^3$.

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?

Test for A[s][i] ...
Program 19.3  Warshall's algorithm

The constructor for class GraphTC computes the transitive closure of G in the private data field T so that clients can use GraphTC objects to test whether any given vertex in a digraph is reachable from any other given vertex. The constructor initializes T with a copy of G, adds self-loops, then uses Warshall's algorithm to complete the computation. We use a DenseGraph object for the transitive closure T because the algorithm needs an efficient implementation of the edge existence test (see Section 17.5).

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

        boolean reachable(int s, int t)
        { return T.edge(s, t); }
    }
}

Table 19.1  Empirical study of transitive-closure algorithms

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

<table>
<thead>
<tr>
<th>V</th>
<th>W</th>
<th>W*</th>
<th>A</th>
<th>L</th>
<th>E</th>
<th>W</th>
<th>W*</th>
<th>A</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>5000</td>
<td>289</td>
<td>203</td>
<td>177</td>
<td>23</td>
</tr>
<tr>
<td>50</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>10000</td>
<td>303</td>
<td>214</td>
<td>184</td>
<td>38</td>
</tr>
<tr>
<td>125</td>
<td>36</td>
<td>21</td>
<td>23</td>
<td>4</td>
<td>26000</td>
<td>309</td>
<td>226</td>
<td>200</td>
<td>97</td>
</tr>
<tr>
<td>250</td>
<td>275</td>
<td>181</td>
<td>178</td>
<td>13</td>
<td>50000</td>
<td>315</td>
<td>232</td>
<td>218</td>
<td>337</td>
</tr>
<tr>
<td>500</td>
<td>2022</td>
<td>1438</td>
<td>1481</td>
<td>64</td>
<td>100000</td>
<td>328</td>
<td>246</td>
<td>235</td>
<td>784</td>
</tr>
</tbody>
</table>

Key:
- W  Warshall's algorithm (Section 19.3)
- W* Improved Warshall's algorithm (Program 19.3)
- A  DFS, adjacency-list representation (Programs 19.4 and 17.7)
- L  DFS, adjacency-lists representation (Program 19.4, 17.9)
Public datasets for *H. sapiens*

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

MCL clustering algorithm

Stijn van Dongen

• Markov (Chain Monte Carlo) Clustering

• Random walks according to edge weights

• Follow the different paths according to their probability

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

With this, the MCL algorithm can be written as:

\[ G \text{ is a graph} \]
\[ \text{add loops to } G \quad \text{# see below} \]
\[ \text{set } K \text{ to some value} \quad \text{# affects granularity} \]
\[ \text{set } M_1 \text{ to be the matrix of random walks on } G \]
\[
\text{while (change) } \\
\quad \text{if } K_2 = K_1 \cdot K_1 \quad \text{# expansion} \\
\quad \text{if } T_2 = T_1 \cdot T_1 \quad \text{# inflation} \\
\quad \text{change } = \text{difference}(M_1, M_2) \\
\]
\[ \text{set CLUSTERING as the components of } M_1 \quad \text{# see below} \]
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 $E$. 

Max-Flow Min-Cut Theorem (Ford Fulkerson's Algorithm)
What is Network Flow?

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

Following is an illustration of a network flow:

![Network Flow Diagram]

Conditions for Network Flow

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

• Capacity Constraint: \(\forall u,v \in V, \ f(u,v) \leq c(u,v)\)
• Skew Symmetry: \(\forall u,v \in V, \ f(u,v) = -f(v,u)\)
• Flow Conservation: \(\forall u \in V - \{s,t\} \sum_{v \in V} f(s,v)=0\)

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 the graph above:

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

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

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

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

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

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

Cuts of Flow Networks

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

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

Min Cut

Min s-t cut (Also called as a Min Cut) is a cut of minimum capacity.
The vertices in S are colored in green.
The vertices in T are colored in grey.
The edges from S to T whose sum of capacity is minimum are colored in pink.

Mayank Joshi, 1/27/2008
Flow of Min Cut (Weak Duality)

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

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

Proof:

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

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

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

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

Methods

Max-Flow Min-Cut Theorem

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

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

\[
rc_{ij} = c_{ij} - f_{ij}
\]

where $c_{ij}$ is a forward edge, and equals $f_{ij}$ when $(i,j)$ is a backward edge. Moreover, the residual capacity of an edge is always non-negative.

\[
c_{ij}(u,v) = c_{ij}(u,v) - f_{ij}(u,v)
\]

The Ford-Fulkerson Method

Begin

\[x := 0; \quad \text{// $x$ is the flow.}\]
create the residual network $G(x)$;
while there is some directed path from $s$ to $t$ in $G(x)$ do
begin
\[\Delta := \delta(P);\]
send $\Delta$ units of flow along $P$;
update the $r$'s;
end
end \{the flow $x$ is now maximum\}.

Click To See Ford Fulkerson’s Algorithm In Action (Animation)
Augmenting Paths (A Useful Concept)

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

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

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

The Ford-Fulkerson’s Algorithm

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

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

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

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

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

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

When is the flow optimal?

A flow \( f \) is maximum flow in G if:

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

**Proof:**

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

2. Let \( v=Fx(S,T) \) 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

This is the original network, plus reversals of the arcs.
Ford-Fulkerson Max Flow

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

Find any s-t path in G(x)
Determine the capacity $\Delta$ of the path.

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

Find any s-t path
Determine the capacity $\Delta$ of the path.

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

Find any s-t path
Determine the capacity $\Delta$ of the path.

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

Find any s-t path
Determine the capacity $\Delta$ of the path.

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

Find any s-t path
Ford-Fulkerson Max Flow

Determine the capacity $\Delta$ of the path.

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

There is no s-t path in the residual network. This flow is optimal.
Ford-Fulkerson Max Flow

These are the nodes that are reachable from node s.

Ford-Fulkerson Max Flow

Here is the optimal flow
Converting Matching to Network Flow