Outline

Today and next lecture:

- Graph Terminology
- Graph representation: adjacency matrix and adjacency list
- Graph traversal: DFS and BFS
- Topological sort
- Minimum Spanning Tree (MST): Prim’s and Kruskal’s algorithms
- Dijkstra’s SPF
Graphs

What is a graph?

Examples of graph: airline flight map, the Internet, the Web, NPC (non-player character) state machine, protocol state machine, object inheritance, makefile dependencies, a program’s call graph, register allocation, data-flow analysis, garbage collection, manufacturing process, organizational cash flow, facility location, etc.

The most versatile of data structures
Graph: Formal Definition

A graph $G = (V, E)$ is a set of vertices (or nodes), $V = \{v_1, v_2, v_3, \ldots\}$, together with a set of edges (or links), $E = \{e_1, e_2, e_3, \ldots\}$, that connect pairs of vertices.

Two vertices are directly connected if there is an edge connecting them.

Directly connected vertices are said to be adjacent to each other, and one is said to be the neighbor of the other.

The edge directly connecting two vertices are said to be incident to the vertices, and the vertices incident to the edge.
Nodes, Edges, Paths

Two vertices may be directly connected by more than one parallel edges.

An edge connecting a vertex to itself is called a self-loop.

A simple graph is a graph without parallel edges and self-loops.

Unless otherwise specified, we will work mostly with simple graph.

A path in $G$ from node $u$ to $v$ is a sequence of vertices from $u$ to $v$ in $G$.

A simple path is a path with no vertex appearing twice.

A connected graph is a graph where a simple path exists between all pairs of vertices.
Weighted Graphs

**Weighted** graph: edges of a graph may have different costs or weights

Example: to go from $A$ to $B$, one can fly, drive on freeway, drive on highway, or bike, or walk on a trail, each has a different cost (in time, money, etc.)

A network connection between two hosts can be through a 24Kbps GPRS, 56Kbps modem, 256Kbps satellite, 256Kbps ADSL, 1Mbps cable modem, 1.5Mbps T1, 2Mbps microwave, 1-3Mbps fixed wireless, 10 Mbps WiFi, 45Mbps T3, 50 Mbps 802.11[ag], 100 Mbps MIMO, 155Mbps OC3, 622Mbps OC12, 1Gbps Ethernet, 10Gbps Ethernet, etc.

**Unweighted** graph: all edges have the same cost
Directed Graphs

Directed graph (digraph):

- edges are directional
- nodes *incident* to an edge form an ordered pair
  - order of nodes on edges is important
  - $e = (u, v)$ means there is an edge from $u$ to $v$, i.e., a path can form from $u$ to $v$ but not vice versa
- examples: rivers and streams, one-way streets, customer-provider relationships
Undirected Graphs

**Undirected** graph:

- there’s no ordering of nodes on edges
- \( e = (u, v) \) means there is an edge *between* \( u \) and \( v \)
  a path can go in either direction
- examples: co-authorship of books, co-starring of movies, team-mates, or any other kinds of *inherently* two-ways, *symmetric* relationships

Sugih Jamin (jamin@eecs.umich.edu)
Node Degree and Connectedness

The **degree** of a node is the number of its (directly connected) neighbors.

For directed edges, we differentiate between **ingress (incoming)** edge and **egress (outgoing)** edge of a node.

Thus we also speak of a node’s **in-degree** and **out-degree**.

A directed graph is **strongly connected** if there is a simple directed path between any pair of vertices.

A directed graph is **weakly connected** if there is a simple path between any pair of vertices in the underlying undirected graph.
Acyclic Graph

A **cycle** is a path starting and finishing at the same node.

A **self-loop** is a cycle of length 1.

A **simple cycle** has no repeated nodes (except the first == last node).

A **simple graph** is a graph with no self-loop (nor parallel edges).

An **acyclic graph** is a graph with no cycle.

A **DAG** is a directed acyclic graph.
Graph Size and Subgraph

The size of a graph, and the complexity of a graph-theoretic algorithm, is usually defined in terms of number of edges $|E|$, number of vertices $|V|$, or both

**Sparse graph**: a graph with few edges, $|E| \ll |V|^2$ or $|E| \approx |V|$

**Dense graph**: a graph with many edges, $|E| \approx |V|^2$

$G' = (V', E')$ is a subgraph of $G = (V, E)$ iff $V' \subset V$ and $E' \subset E$

A **clique** is a subgraph where every node pair is directly connected

A **complete graph** (or (full-)mesh) is a graph where every node pair is directly connected

How many edges are there in a complete graph of $N$ nodes?
Graph Algorithm Questions

See airline flight graph of GTM Fig. 12.14

Describe an algorithm to determine:

- all non-stop flights from JFK \( (O( )) \)
- if any non-stop flights from JFK exist \( (O( )) \)
- greatest distance covered by a non-stop flight from JFK \( (O( )) \)

Associate a price with each flight, describe an algorithm to determine:

- best “value” non-stop flight from JFK (max distance/price) \( (O( )) \)
- best “value” tour (non-stop or connecting flights) from JFK \( (O( )) \)
Graph Algorithm Questions

Describe an algorithm to determine:

- if a flight from JFK to SFO exists \( (O(\quad)) \)
- minimal cost (distance or price) from JFK to SFO \( (O(\quad)) \)

Suppose numbers represent cost (in billions USD) to build high-speed rail, describe an algorithm to determine least cost construction, such that any city can be reached from any other city \( (O(\quad)) \)

Suppose you are planning a family reunion. Your family is spread out all over the US and you are paying for their travel. Describe an algorithm to find the city to host the reunion that minimizes total travel cost \( (O(\quad)) \)
Graph Representation: Adjacency Matrix

The graph:

\[
\begin{array}{cccccccc}
\text{a} & \text{b} & \text{c} & \text{d} & \text{e} & \text{f} & \text{g} \\
\hline
\text{a} & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\
\text{b} & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
\text{c} & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\
\text{d} & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
\text{e} & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
\text{f} & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\
\text{g} & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
\]

Adjacency matrix:

- an edge in an unweighted graph is represented with a ‘1’, ‘0’ otherwise
- only needs \( O(|V|^2) \) bits to represent an unweighted graph
- an edge in a weighted graph is represented with its weight, ‘\( \infty \)’ otherwise
- undirected graph only needs \( O(|V|^2/2) \) space

Sugih Jamin (jamin@eecs.umich.edu)
Graph Representation: Adjacency List

The graph:

```
a
  |   |
  |   |
  +---+---+
     
  c
  |
  |
  f
```

as an adjacency list:

```
<table>
<thead>
<tr>
<th>a</th>
<th>c</th>
<th>d</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>d</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>a</td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>a</td>
<td>b</td>
<td>e</td>
</tr>
<tr>
<td>e</td>
<td>b</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>a</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Adjacency list:

- can also be implemented as linked list
- weighted graph stores edge weight on link-list node
- undirected graph must represent each edge twice
- space used is $O(|V| + |E|)$

Sugih Jamin (jamin@eecs.umich.edu)
Graph Traversal: Depth-first Traversal

Recursively follow a path, visiting each successive node until all neighbors are visited

Can be implemented using a stack

- push start node onto stack
- until stack is empty:
  - pop the stack
  - if node not visited, visit node:
    - mark the popped node visited
    - push all of its neighbors not marked visited onto the stack

Can be used to perform depth-first search (DFS) to discover a path between two nodes (not necessarily the shortest nor a unique path)
Example

- **A** unexplored vertex
- **A** visited vertex
- **A** unexplored edge
- **A** discovery edge
- **A** back edge
Example (cont.)
Graph Traversal: Breadth-first Traversal

Visit all directly connected neighbors first, then nodes 2 hops away, 3 hops away, and so on.

Can be implemented using a queue:

- enqueue start node
- until queue is empty
  - dequeue first element from queue
  - if node not visited, visit node:
    - mark the dequeued node visited
    - enqueue all of its neighbors not marked visited into the queue

Can be used to perform breadth-first search (BFS) to discover the shortest path between a source and destination nodes.

Sugih Jamin (jamin@eecs.umich.edu)
Example

- A: unexplored vertex
- A: visited vertex
- unexplored edge
- discovery edge
- cross edge

Diagram:

- $L_0$: Level 0
- $L_1$: Level 1

Nodes: A, B, C, D, E, F

Connections:
- A to B
- B to C
- C to D
- D to E
- D to F

Arrows indicate the direction of traversal.
Example (cont.)
Example (cont.)
Running Times of Graph Traversal Algorithms

If graph implemented as adjacency matrix:

- both $\text{DFS()}$ and $\text{BFS()}$ “visit” each node once: $O(|V|)$
- the adjacency row of each node is examined once: $O(|V|)$
- total: $O(|V|^2)$

Why bother with adjacency matrix?

If graph implemented as adjacency list:

- both $\text{DFS()}$ and $\text{BFS()}$ “visit” each node once: $O(|V|)$
- each (undirected) edge examined twice: $O(|E|)$
- total: $O(|V| + |E|)$
Makefile Dependencies

Consider this Makefile:

all: A B

A: a.o c.o
   g++ -o A a.o c.o

a.o: a.cpp d.h
    g++ -c a.cpp

c.o: c.cpp
    g++ -c c.cpp

B: b.cpp d.h
   g++ -o B b.cpp

In what ordering should you make the files?
Makefile Implementation: Topological Sort

Perform DAG check and **topological sort** concurrently, do a depth first traversal from **start node**:

```c
int numv = |V|;
int DAGcheck_Toposort(start_node)
{
    mark node VISITING;
    for each neighbor
        if neighbor is not marked VISITING nor VISITED,
            if (! DAGcheck_Toposort(neighbor)) return FALSE;
        else if neighbor is marked VISITING,
            return FALSE; // cycle: dependency graph not a DAG!
    when all neighbors are marked VISITED
        mark node VISITED; number node numv; numv--;
    return TRUE;
}
```

The first node with all neighbors visited will have the largest `numv`

Make the files in increasing `numv`

Running time: \( \Theta(|V| + |E|) \)
Acyclic Graph

A cycle is a path starting and finishing at the same node

A self-loop is a cycle of length 1

A simple cycle has no repeated nodes (except the first == last node)

A simple graph is a graph with no self-loop (nor parallel edges)

An acyclic graph is a graph with no cycle

A DAG is a directed acyclic graph
Graph Size and Subgraph

The size of a graph, and the complexity of a graph-theoretic algorithm, is usually defined in terms of number of edges $|E|$, number of vertices $|V|$, or both

**Sparse graph**: a graph with few edges, $|E| \ll |V|^2$ or $|E| \approx |V|$

**Dense graph**: a graph with many edges, $|E| \approx |V|^2$

$G' = (V', E')$ is a **subgraph** of $G = (V, E)$ iff $V' \subset V$ and $E' \subset E$

A **clique** is a subgraph where every node pair is directly connected

A **complete graph** (or (full-)mesh) is a graph where every node pair is directly connected

How many edges are there in a complete graph of $N$ nodes?
Graph Representation: Adjacency Matrix

The graph:

![Graph Diagram]

as an adjacency matrix:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>f</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>g</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Adjacency matrix:

- an edge in an unweighted graph is represented with a ‘1’, ‘0’ otherwise
- only needs $O(|V|^2)$ bits to represent an unweighted graph
- an edge in a weighted graph is represented with its weight, ‘$\infty$’ otherwise
- undirected graph only needs $O(|V|^2/2)$ space

Sugih Jamin (jamin@eecs.umich.edu)
Graph Representation: Adjacency List

The graph:

```
  a -- c -- d -- f
   |     ^     |
   |     |     |
   v     v     v
  b     d     e
   |     |     |
   |     |     |
   v     v     v
  c     f     g
```

As an adjacency list:

```
a  c  d  f
b  d  e
c  a  f
d  a  b  e  f
e  b  d
f  a  c
g
```

Adjacency list:
- can also be implemented as linked list
- weighted graph stores edge weight on link-list node
- undirected graph must represent each edge twice
- space used is $O(|V| + |E|)$
Graph Traversal: Depth-first Traversal

Recursively follow a path, visiting each successive node until all neighbors are visited

Can be implemented using a stack

- push start node onto stack
- until stack is empty:
  - pop the stack
  - if node not visited, visit node:
    - mark the popped node visited
    - push all of its neighbors not marked visited onto the stack

Can be used to perform depth-first search (DFS) to discover a path between two nodes (not necessarily the shortest nor a unique path)
Example

- Unexplored vertex
- Visited vertex
- Unexplored edge
- Discovery edge
- Back edge
Example (cont.)
Graph Traversal: Breadth-first Traversal

Visit all directly connected neighbors first, then nodes 2 hops away, 3 hops away, and so on

Can be implemented using a queue:

- enqueue start node
- until queue is empty
  - dequeue first element from queue
  - if node not visited, visit node:
    - mark the dequeued node visited
    - enqueue all of its neighbors not marked visited into the queue

Can be used to perform breadth-first search (BFS) to discover the shortest path between a source and destination nodes
Example

- **A**: unexplored vertex
- **B**: visited vertex
- **C**: unexplored edge
- **D**: discovery edge
- **E**: cross edge

$L_0$: Level 0

$L_1$: Level 1
Example (cont.)

Diagram showing a sequence of states and transitions labeled as $L_0$, $L_1$, and $L_2$. The states are represented by circles labeled A, B, C, D, E, and F.
Example (cont.)
Running Times of Graph Traversal Algorithms

If graph implemented as adjacency matrix:

- both \( \text{DFS}() \) and \( \text{BFS}() \) “visit” each node once: \( O(|V|) \)
- the adjacency row of each node is examined once: \( O(|V|) \)
- total: \( O(|V|^2) \)

Why bother with adjacency matrix?

If graph implemented as adjacency list:

- both \( \text{DFS}() \) and \( \text{BFS}() \) “visit” each node once: \( O(|V|) \)
- each (undirected) edge examined twice: \( O(|E|) \)
- total: \( O(|V| + |E|) \)
Consider this Makefile:

all: A B

A: a.o c.o
   g++ -o A a.o c.o

a.o: a.cpp d.h
   g++ -c a.cpp

B: b.cpp d.h
   g++ -o B b.cpp

In what ordering should you make the files?
Makefile Implementation: Topological Sort

Perform DAG check and **topological sort** concurrently, do a depth first traversal from **start node**:

```c
int numv = |V|;
int DAGcheck_Toposort(start_node)
{
    mark node VISITING;
    for each neighbor
        if neighbor is not marked VISITING nor VISITED,
            if (! DAGcheck_Toposort(neighbor)) return FALSE;
        else if neighbor is marked VISITING,
            return FALSE; // cycle: dependency graph not a DAG!
    when all neighbors are marked VISITED
        mark node VISITED; number node numv; numv--;
    return TRUE;
}
```

The first node with all neighbors visited will have the largest numv
Make the files in increasing numv

Running time: $\Theta(|V| + |E|)$
Subgraph and Spanning Tree

$G' = (V', E')$ is a subgraph of $G = (V, E)$ iff $V' \subset V$ and $E' \subset E$

A spanning tree is a minimal subgraph of a connected graph. $T = (V', E')$ is a spanning tree of $G = (V, E)$ iff

- $V' = V$ (T spans G)
- $T$ is a tree:
  - $T$ is connected
  - $T$ is acyclic

If $T = (V, E')$ is a spanning tree of $G = (V, E)$, then if $|V| > 1$, $T$ contains at least one vertex of degree 1; and since $T$ is a tree, $|E'| = |V| - 1$

The results of both a DFS and a BFS are spanning trees, but not a minimum spanning tree

Sugih Jamin (jamin@eecs.umich.edu)
Spanning Trees

\[ G: \]

\[ \text{BFS:} \]

\[ \text{DFS:} \]

\[ \text{MST:} \]
Minimum Spanning Tree

**Minimum Spanning Tree:**
Given an edge-weighted, connected, undirected graph $G = (V, E)$, a minimum spanning tree $T$ spans (contains all nodes in) $G$ and the sum of the costs of all edges in $T$ is minimal

$C(e)$: cost/weight associated with an edge

Total cost of tree $T = \sum_{e \in T} C(e)$

Minimum spanning tree (MST):
find $T = (V, E')$ such that $\sum_{e \in E'} C(e)$ is minimal
Example Uses of MSTs

- DoT planning the federal freeway system
- A nationwide delivery company planning its truck route
- An airline planning its flight paths (hub-and-spoke system or feeder system)
- A railroad company planning where to lay down tracks
- A power company planning where to lay down high-voltage power lines
- A network provider planning where to lay down trunk fibers for its backbone network
- A peer-to-peer network forming a multicast delivery tree
Prim’s Algorithm

A greedy algorithm for finding MST:

1. given $G = (V, E)$ a weighted, connected, undirected graph
2. separate $V$ into two sets:
   - $T$: nodes on the MST
   - $T^c$: those not
3. $T$ initially empty, choose a random node and add it to $T$
4. select an edge with the smallest cost/weight/distance from any node in $T$ that connects to a node $v$ in $T^c$, move $v$ to $T$
5. repeat step 4 until $T^c$ is empty
Prim's Algorithm

Prim(startnode s)
{
    // Initialize
    table = createtable(|V|); // stores m, c, p
    table[*].mst = false; table[*].cost = INFINITY;
    pq = createpq(|E|); // empty pq
    table[s].cost = 0;
    pq.insert(0, s); // pq.insert(d, v)

    while (!pq.isempty()) {
        v = pq.getMin();
        if (!table[v].mst) { // not on T
            table[v].mst = true;
            for each u = v.neighbors() {
                newcost = weight(u, v);
                if (table[u].cost > newcost) {
                    table[u].cost = newcost;
                    table[u].predecessor = v;
                    pq.insert(newcost, u);
                }
            }
        }
    }
    extract MST from table;
}
### Prim’s Example (init)

<table>
<thead>
<tr>
<th>$u$</th>
<th>mst</th>
<th>cost</th>
<th>pred</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>F</td>
<td>0</td>
<td>$-$</td>
</tr>
<tr>
<td>$b$</td>
<td>F</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>$c$</td>
<td>F</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>$d$</td>
<td>F</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>$e$</td>
<td>F</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>$f$</td>
<td>F</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

![Graph](image)
Prim’s Example (a)

<table>
<thead>
<tr>
<th>$u$</th>
<th>mst</th>
<th>cost</th>
<th>pred</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>T</td>
<td>0</td>
<td>–</td>
</tr>
<tr>
<td>$b$</td>
<td>F</td>
<td>13</td>
<td>$a$</td>
</tr>
<tr>
<td>$c$</td>
<td>F</td>
<td>8</td>
<td>$a$</td>
</tr>
<tr>
<td>$d$</td>
<td>F</td>
<td>1</td>
<td>$a$</td>
</tr>
<tr>
<td>$e$</td>
<td>F</td>
<td>$\infty$</td>
<td></td>
</tr>
<tr>
<td>$f$</td>
<td>F</td>
<td>$\infty$</td>
<td></td>
</tr>
</tbody>
</table>

Sugih Jamin (jamin@eecs.umich.edu)
### Prim’s Example (d)

<table>
<thead>
<tr>
<th>( u )</th>
<th>mst</th>
<th>cost</th>
<th>pred</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>T</td>
<td>0</td>
<td>–</td>
</tr>
<tr>
<td>( b )</td>
<td>F</td>
<td>13</td>
<td>( a )</td>
</tr>
<tr>
<td>( c )</td>
<td>F</td>
<td>5</td>
<td>( d )</td>
</tr>
<tr>
<td>( d )</td>
<td>T</td>
<td>1</td>
<td>( a )</td>
</tr>
<tr>
<td>( e )</td>
<td>F</td>
<td>4</td>
<td>( d )</td>
</tr>
<tr>
<td>( f )</td>
<td>F</td>
<td>5</td>
<td>( d )</td>
</tr>
</tbody>
</table>

Sugih Jamin (jamin@eecs.umich.edu)
### Prim's Example (e)

<table>
<thead>
<tr>
<th>$u$</th>
<th>mst</th>
<th>cost</th>
<th>pred</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>T</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$b$</td>
<td>F</td>
<td>13</td>
<td>$a$</td>
</tr>
<tr>
<td>$c$</td>
<td>F</td>
<td>3</td>
<td>$e$</td>
</tr>
<tr>
<td>$d$</td>
<td>T</td>
<td>1</td>
<td>$a$</td>
</tr>
<tr>
<td>$e$</td>
<td>T</td>
<td>4</td>
<td>$d$</td>
</tr>
<tr>
<td>$f$</td>
<td>F</td>
<td>2</td>
<td>$e$</td>
</tr>
</tbody>
</table>
Prim’s Example (f)

<table>
<thead>
<tr>
<th>$u$</th>
<th>mst</th>
<th>cost</th>
<th>pred</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>T</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>$b$</td>
<td>F</td>
<td>13</td>
<td>$a$</td>
</tr>
<tr>
<td>$c$</td>
<td>F</td>
<td>3</td>
<td>$e$</td>
</tr>
<tr>
<td>$d$</td>
<td>T</td>
<td>1</td>
<td>$a$</td>
</tr>
<tr>
<td>$e$</td>
<td>T</td>
<td>4</td>
<td>$d$</td>
</tr>
<tr>
<td>$f$</td>
<td>T</td>
<td>2</td>
<td>$e$</td>
</tr>
</tbody>
</table>
### Prim's Example (c)

<table>
<thead>
<tr>
<th>( u )</th>
<th>mst</th>
<th>cost</th>
<th>pred</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>T</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>( b )</td>
<td>F</td>
<td>13</td>
<td>( a )</td>
</tr>
<tr>
<td>( c )</td>
<td>T</td>
<td>3</td>
<td>( e )</td>
</tr>
<tr>
<td>( d )</td>
<td>T</td>
<td>1</td>
<td>( a )</td>
</tr>
<tr>
<td>( e )</td>
<td>T</td>
<td>4</td>
<td>( d )</td>
</tr>
<tr>
<td>( f )</td>
<td>T</td>
<td>2</td>
<td>( e )</td>
</tr>
</tbody>
</table>
### Prim's Example (b)

<table>
<thead>
<tr>
<th>$u$</th>
<th>mst</th>
<th>cost</th>
<th>pred</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>T</td>
<td>0</td>
<td>—</td>
</tr>
<tr>
<td>$b$</td>
<td>T</td>
<td>13</td>
<td>$a$</td>
</tr>
<tr>
<td>$c$</td>
<td>T</td>
<td>3</td>
<td>$e$</td>
</tr>
<tr>
<td>$d$</td>
<td>T</td>
<td>1</td>
<td>$a$</td>
</tr>
<tr>
<td>$e$</td>
<td>T</td>
<td>4</td>
<td>$d$</td>
</tr>
<tr>
<td>$f$</td>
<td>T</td>
<td>2</td>
<td>$e$</td>
</tr>
</tbody>
</table>
Prim’s Running Time Complexity

Each node must go thru the minHeap at least once: $O(|V| \log |V|)$

Each neighbor of each node could also potentially go thru the minHeap once: $O(|E| \log |V|)$

Total: $O(|V| \log(|V|) + |E| \log(|V|)) = O(|E| \log |V|)$

($|E| \geq |V| - 1$ for a connected graph)

If the pq is implemented as a Fibonacci Heap instead of a binary minHeap, Prim’s algorithm takes $O(|E| + |V| \log |V|)$
Kruskal’s Algorithm

Connected nodes are said to form a **partition**
An edge causes a cycle if it connects nodes already in the same partition
An edge connecting nodes from 2 partition causes them to merge into 1

Kruskal’s algorithm is also a greedy algorithm:

1. given $G = (V, E)$ a weighted, connected, undirected graph
2. create two sets of edges: $F$ and $F^c$
3. initially all edges are in $F^c$, sorted by weight
4. select the smallest cost edge in $F^c$ that does not cause a cycle in $F$ and add it to $F$
5. repeat step 4 until $T = (V, F)$ is a tree
6. $T$ is an MST

Sugih Jamin (jamin@eecs.umich.edu)
Kruskal's Example
Kruskal’s Example

Sugih Jamin (jamin@eecs.umich.edu)
Kruskal's Example
Kruskal’s Example

Sugih Jamin (jamin@eecs.umich.edu)
Kruskal’s Example

Sugih Jamin (jamin@eecs.umich.edu)
Kruskal’s Running Time Complexity

Cycle detection and building a tree from edges require union-find operations which are not simple to implement, but takes time \( O(|E| \log |V|) \)

Kruskal’s algorithm also depends on the sorting algorithm, which takes \( O(|E| \log |E|) \) time

Total time is \( O(|E| \log |E|) = O(|E| \log |V|) \) since in the worst case, \( |E| < |V|^2 \)

So, use Prim’s, it’s simpler to implement

Sugih Jamin (jamin@eecs.umich.edu)
Shortest Path

Example uses:
- minimize travel distance
- minimize travel time
- minimize travel fares
- minimize number of stop-lights

Which path will you take from umich.edu to movies.com?

Sugih Jamin (jamin@eecs.umich.edu)
Single-source Shortest Path

Let $G = (V, E)$ be a weighted, connected, directed graph, $P = \{v_1, v_2, v_3, \ldots, v_k\}$ a path in $G$, and $C(v_i, v_j)$ the weight/cost on the edge connecting $v_i$ to $v_j$.

The **weighted path length** of $P = \sum_{i=1}^{k-1} C(v_i, v_{i+1})$.

Single-source (source-specific, source-rooted) shortest path problem:
Given a weighted, connected, directed graph $G = (V, E)$ and a vertex $s \in V$, find the shortest (smallest weighted) path length from $s$ to all other vertices in $V$.
Weighted vs. Unweighted Graph

Shortest weighted path from $b$ to $f$: $\{b, a, c, e, f\}$

Shortest unweighted path from $b$ to $f$: $\{b, c, e, f\}$
Positive Weight

Shortest path problem **undefined** for graphs with negative-cost cycles:

- Cost of \{d, a, c, e, f\}: 4
- Cost of \{d, a, c, d, a, c, e, f\}: 2
- Cost of \{d, a, c, d, a, c, d, a, c, e, f\}: 0
Dijkstra’s Shortest Path First (SPF) Algorithm

A greedy algorithm for solving single-source shortest path problem

- assume non-negative edge weights
- even if we’re only interested in the path from $s$ to a single destination vertex $d$, we need to find the shortest path from $s$ to all vertices in $G$ (otherwise, we might have missed a shorter path)
- if the shortest path from $s$ to $d$ passes through an intermediate node $u$, i.e., $P = \{s, \ldots, u, \ldots, d\}$, then $P' = \{s, \ldots, u\}$ must be the shortest path from $s$ to $u$
- very similar to Prim’s MST algorithm, with the same running time complexity $O(|E| \log |V|)$ (or $O(|E| + |V| \log |V|)$ if implemented using Fibonacci Heap)
- whereas for Prim’s, for each node $u$ we only need to know the lowest cost edge incident to $u$, for Dijkstra’s we need to know the lowest cost path from $s$ to $u$

Sugih Jamin (jamin@eecs.umich.edu)
Dijkstra's SPF Algorithm

Dijkstra(startnode s)
{
    // Initialize
    table = createtable(|V|); // stores spf, cost, predecessor
    table[*].spf = false; table[*].cost = INFINITY;
    pq = createpq(|E|); // empty pq
    table[s].cost = 0;
    pq.insert(0, s); // pq.insert(d, v)

    while (!pq.isempty()) {
        v = pq.getMin();
        if (!table[v].spf) { // not on T
            table[v].spf = true;
            for each u = v.neighbors() {
                newcost = weight(u, v) + table[v].cost; // diff from Prim's
                if (table[u].cost > newcost) {
                    table[u].cost = newcost;
                    table[u].pred = v;
                    pq.insert(newcost, u);
                }
            }
        }
    }
    extract SPF from table;
}
## Dijsktra’s SPF Example (init)

<table>
<thead>
<tr>
<th>$u$</th>
<th>spf</th>
<th>cost</th>
<th>pred</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>F</td>
<td>$\infty$</td>
<td></td>
</tr>
<tr>
<td>$b$</td>
<td>F</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>$c$</td>
<td>F</td>
<td>$\infty$</td>
<td></td>
</tr>
<tr>
<td>$d$</td>
<td>F</td>
<td>$\infty$</td>
<td></td>
</tr>
<tr>
<td>$e$</td>
<td>F</td>
<td>$\infty$</td>
<td></td>
</tr>
<tr>
<td>$f$</td>
<td>F</td>
<td>$\infty$</td>
<td></td>
</tr>
</tbody>
</table>

The diagram shows the network with edge weights:

- Edge $(a, b)$: 3
- Edge $(a, c)$: 1
- Edge $(b, c)$: 2
- Edge $(b, d)$: 5
- Edge $(c, e)$: 4
- Edge $(d, f)$: 5

Sugih Jamin (jamin@eecs.umich.edu)
Dijsktra’s SPF Example (b)

<table>
<thead>
<tr>
<th>$u$</th>
<th>spf</th>
<th>cost</th>
<th>pred</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>F</td>
<td>3</td>
<td>b</td>
</tr>
<tr>
<td>$b$</td>
<td>T</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>$c$</td>
<td>F</td>
<td>5</td>
<td>b</td>
</tr>
<tr>
<td>$d$</td>
<td>F</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>$e$</td>
<td>F</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>$f$</td>
<td>F</td>
<td>∞</td>
<td></td>
</tr>
</tbody>
</table>
Dijsktra’s SPF Example (a)

<table>
<thead>
<tr>
<th>u</th>
<th>spf</th>
<th>cost</th>
<th>pred</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>T</td>
<td>3</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>T</td>
<td>0</td>
<td>–</td>
</tr>
<tr>
<td>c</td>
<td>F</td>
<td>4</td>
<td>a</td>
</tr>
<tr>
<td>d</td>
<td>F</td>
<td>8</td>
<td>a</td>
</tr>
<tr>
<td>e</td>
<td>F</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>F</td>
<td>∞</td>
<td></td>
</tr>
</tbody>
</table>

Sugih Jamin (jamin@eecs.umich.edu)
Dijsktra’s SPF Example (c)

<table>
<thead>
<tr>
<th>$u$</th>
<th>spf</th>
<th>cost</th>
<th>pred</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>T</td>
<td>3</td>
<td>b</td>
</tr>
<tr>
<td>$b$</td>
<td>T</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>$c$</td>
<td>T</td>
<td>4</td>
<td>a</td>
</tr>
<tr>
<td>$d$</td>
<td>F</td>
<td>6</td>
<td>c</td>
</tr>
<tr>
<td>$e$</td>
<td>F</td>
<td>8</td>
<td>c</td>
</tr>
<tr>
<td>$f$</td>
<td>F</td>
<td>$\infty$</td>
<td></td>
</tr>
</tbody>
</table>

Sugih Jamin (jamin@eecs.umich.edu)
Dijsktra’s SPF Example (d)

<table>
<thead>
<tr>
<th>( u )</th>
<th>spf</th>
<th>cost</th>
<th>pred</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>T</td>
<td>3</td>
<td>b</td>
</tr>
<tr>
<td>( b )</td>
<td>T</td>
<td>0</td>
<td>–</td>
</tr>
<tr>
<td>( c )</td>
<td>T</td>
<td>4</td>
<td>a</td>
</tr>
<tr>
<td>( d )</td>
<td>T</td>
<td>6</td>
<td>c</td>
</tr>
<tr>
<td>( e )</td>
<td>F</td>
<td>8</td>
<td>c</td>
</tr>
<tr>
<td>( f )</td>
<td>F</td>
<td>11</td>
<td>d</td>
</tr>
</tbody>
</table>

Sugih Jamin (jamin@eecs.umich.edu)
### Dijsktra’s SPF Example (e)

<table>
<thead>
<tr>
<th>$u$</th>
<th>spf</th>
<th>cost</th>
<th>pred</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>T</td>
<td>3</td>
<td>b</td>
</tr>
<tr>
<td>$b$</td>
<td>T</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>$c$</td>
<td>T</td>
<td>4</td>
<td>a</td>
</tr>
<tr>
<td>$d$</td>
<td>T</td>
<td>6</td>
<td>c</td>
</tr>
<tr>
<td>$e$</td>
<td>T</td>
<td>8</td>
<td>c</td>
</tr>
<tr>
<td>$f$</td>
<td>F</td>
<td>9</td>
<td>e</td>
</tr>
</tbody>
</table>

Sugih Jamin (jamin@eecs.umich.edu)
Dijsktra’s SPF Example (f)

<table>
<thead>
<tr>
<th>$u$</th>
<th>spf</th>
<th>cost</th>
<th>pred</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>T</td>
<td>3</td>
<td>b</td>
</tr>
<tr>
<td>$b$</td>
<td>T</td>
<td>0</td>
<td>–</td>
</tr>
<tr>
<td>$c$</td>
<td>T</td>
<td>4</td>
<td>a</td>
</tr>
<tr>
<td>$d$</td>
<td>T</td>
<td>6</td>
<td>c</td>
</tr>
<tr>
<td>$e$</td>
<td>T</td>
<td>8</td>
<td>c</td>
</tr>
<tr>
<td>$f$</td>
<td>T</td>
<td>9</td>
<td>e</td>
</tr>
</tbody>
</table>