Graphs

Undirected

Directed

Shelton (UC Riverside)
Graphs

undirected

directed
vertex  item in the graph (also called a node)
edge  connection from one vertex to another (also called an arc)
    directed  connection has “direction” (an arrow)
    undirected  connection is symmetric (no arrow)
    weight  connection might have a weight (cost, distance, etc.)
directed graph  a graph with directed edges
undirected graph  a graph with undirected edges
Graph Notation

path  a sequence of vertices, each connected by an edge
  simple  a path is simple if it doesn’t repeat a vertex

cycle  a path (of length $> 0$) that starts and ends at the same point
  undirected  if the graph is undirected, it cannot repeat an edge

acyclic  a (directed) graph that has no cycles (also called a DAG)
  connected  a connected graph has a path from every vertex to every other
  strongly connected  same as connected, but for a directed graph
  weakly connected  a directed graph is not strongly connected, would would
                  be connected if all edges were undirected

complete  a complete graph has all possible edges
Graph Examples

social networks Each person is a node, each edge is a friendship

economic trade Each country/company/etc. is a node, each edge is an economic transaction

roads Each intersection is a node, each edge is a road (segment)

airlines Each intersection is an airport, each edge is a flight
Graph Data Structure

$V$ is the set of vertices, $E$ is the set of edges
Usually let $V = \{0, 1, 2, \ldots, n - 1\}$.

How to represent?

- **adjacency matrix**: a 2D array of booleans
  
  ```
  vector<vector<bool>> adj;
  // adj[i][j] is true if edge from i to j
  ```

- **adjacency list**: a 1D array of lists of adjacent vertices
  
  ```
  vector<vector<int>> adj;
  // adj[i][k] == j => an edge from i to j
  ```

If the graph is dense (each vertex has $O(|V|)$ adjacent nodes), either is okay.
If the graph is sparse (each vertex has few adjacent nodes), the second is much better.
Graph Data Structure

$V$ is the set of vertices, $E$ is the set of edges

Usually let $V = \{0, 1, 2, \ldots, n - 1\}$.

How to represent?

**adjacency matrix** a 2D array of booleans

```cpp
vector<vector<bool>> adj;
// adj[i][j] is true if edge from i to j
```
Graph Data Structure

$V$ is the set of vertices, $E$ is the set of edges
Usually let $V = \{0, 1, 2, \ldots, n - 1\}$.

How to represent?

- **adjacency matrix** a 2D array of booleans
  
  \[
  \text{vector<vector<bool>> adj;} \\
  \quad // \text{adj[i][j] is true if edge from i to j}
  \]

- **adjacency list** a 1D array of lists of adjacent vertices
  
  \[
  \text{vector<vector<int>> adj;} \\
  \quad // \text{adj[i][k] == j => an edge from i to j}
  \]
Graph Data Structure

\( V \) is the set of vertices, \( E \) is the set of edges

Usually let \( V = \{0, 1, 2, \ldots, n - 1\} \).

How to represent?

**adjacency matrix** a 2D array of booleans

```cpp
vector<vector<bool>> adj;
// adj[i][j] is true if edge from i to j
```

**adjacency list** a 1D array of lists of adjacent vertices

```cpp
vector<vector<int>> adj;
// adj[i][k] == j => an edge from i to j
```

If the graph is dense (each vertex has \( O(|V|) \) adjacent nodes),
either is okay

If graph is sparse (each vertex has few adjacent nodes),
second is much better
Graph Data Structure

Adjacency Matrix

\[
\begin{array}{ccccccc}
& A & B & C & D & E & F \\
A & 0 & 0 & 0 & 1 & 0 & 0 \\
B & 0 & 0 & 1 & 0 & 1 & 1 \\
C & 0 & 1 & 0 & 1 & 1 & 0 \\
D & 1 & 1 & 0 & 0 & 1 & 0 \\
E & 1 & 0 & 0 & 0 & 0 & 0 \\
F & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]
Adjacency List

A  D
B  C, E, F
C  D, B, E
D  A, B, E
E  A
F  

Graph Data Structure
Topological Sort

CS10
CS12
CS14
CS61
CS100
CS141
CS153
CS152
CS111
CS11
CS150
CS120A
CS120B
CS161
MATH9A
MATH9B
MATH9C
CS152
CS120B
CS161
Graphs I

Shelton (UC Riverside)
Topological Sort

Put a graph in an order such that if there is a path from $i$ to $j$, $i$ appears before $j$.

Only possible if graph is acyclic.
Topological Sort

Put a graph in an order such that if there is a path from $i$ to $j$, $i$ appears before $j$.

Only possible if graph is acyclic.

How?

class graph {
    // ... assume normal functions to add nodes and edges
    vector<int> toposort() const {
    }

private:
    vector<vector<int>> adj; // adjacency list
};
Topological Sort

Idea: indegree (number of inbound edges)

- Take nodes with indegree 0.
- Decrease the indegree of each nodes adjacent.
- Repeat
Topological Sort

Idea: indegree (number of inbound edges)

```cpp
vector<int> toposort() const {
    vector<int> ret;
    vector<int> indegrees(adj.size(),0);
    for(auto &alst : adj)
        for(int j : alst) indegree[j]++;

    for(int i=0;i<adj.size();i++) {
        int j = finddegzero(indegrees);
        ret.push_back(j);
        for(int k : adj[j]) indegree[k]--;
        indegree[j] = -1; // -1 = "done"
    }
    return ret;
}
```
Topological Sort

CS10, MATH9A, CS12, CS11, CS111, MATH9B

output:

CS10 0
CS11 1
CS12 2
CS111 2
Math9A 0
Math9B 1
output: CS10
output: CS10, MATH9A
Topological Sort

output: CS10, MATH9A, CS12
output: CS10, MATH9A, CS12, CS11
output: CS10, MATH9A, CS12, CS11, CS111
output: CS10, MATH9A, CS12, CS11, CS111, MATH9B
vector<int> toposort() const {
    vector<int> ret;
    vector<int> indegrees(adj.size(), 0);
    for (auto &alst : adj)
        for (int j : alst) indegree[j]++;

    for (int i = 0; i < adj.size(); i++) {
        int j = finddegzero(indegrees);
        ret.push_back(j);
        for (int k : adj[j]) indegree[k]--;
        indegree[j] = -1; // -1 = "done"
    }
    return ret;
}
Topological Sort

vector<int> toposort() const {
    vector<int> ret;
    vector<int> indegrees(adj.size(),0);
    for(auto &alst : adj)
        for(int j : alst) indegree[j]++;
    for(int i=0;i<adj.size();i++) {
        int j = finddegzero(indegrees);
        ret.push_back(j);
        for(int k : adj[j]) indegree[k]--;
        indegree[j] = -1; // -1 = "done"
    }
    return ret;
}

Running time? $O(|V|^2)$
vector<int> toposort() const {
    vector<int> ret;
    vector<int> indegrees(adj.size(),0);
    for(auto &alst : adj)
        for(int j : alst) indegree[j]++;

    for(int i=0;i<adj.size();i++) {
        int j = finddegzero(indegrees);
        ret.push_back(j);
        for(int k : adj[j]) indegree[k]--;
        indegree[j] = -1; // -1 = "done"
    }
    return ret;
}
vector<int> toposort() const {
    vector<int> ret;
    vector<int> indegrees(adj.size(), 0);
    for (auto &alst : adj) {
        for (int j : alst) indegree[j]++;
    }
    for (int i = 0; i < adj.size(); i++) {
        int j = finddegzero(indegrees);
        ret.push_back(j);
        for (int k : adj[j]) indegree[k]--;
        indegree[j] = -1; // -1 = "done"
    }
    return ret;
}

Running time? \( O(|V|^2) \)

Improve by keeping list/set of 0-indegree nodes

Running time? \( O(|V| + |E|) \)
vector<int> toposort() const {
    vector<int> ret;
    vector<int> indegrees(adj.size(),0);
    for(auto &alst : adj)
        for(int j : alst) indegree[j]++;
    for(int i=0;i<adj.size();i++) {
        int j = finddegzero(indegrees);
        ret.push_back(j);
        for(int k : adj[j]) indegree[k]--;
        indegree[j] = -1; // -1 = "done"
    }
    return ret;
}

Running time? $O(|V|^2)$

Improve by keeping list/set of 0-indegree nodes
Running time? $O(|V| + |E|)$

What if node ids are not 0, 1, ..., $n - 1$?
vector<int> toposort() const {
    vector<int> ret;
    vector<int> indegrees(adj.size(), 0);
    for (auto &alst : adj) {
        for (int j : alst) indegree[j]++;
    }

    for (int i = 0; i < adj.size(); i++) {
        int j = finddegzero(indegrees);
        ret.push_back(j);
        for (int k : adj[j]) indegree[k]--;
        indegree[j] = -1; // -1 = "done"
    }

    return ret;
}

Running time? \(O(|V|^2)\)

Improve by keeping list/set of 0-indegree nodes
Running time? \(O(|V| + |E|)\)

What if node ids are not 0, 1, ..., \(n - 1\)?
- \(\text{adj}\) is of type \(\text{map<nodeT, vector<map<...>::iterator>>}\)
- \(\text{indegrees}\) is of type \(\text{map<nodeT, int>}\)
Goal: find shortest path from C to all other vertices.
Shortest-Path

Goal: find shortest path from C to all other vertices.
Goal: find shortest path from C to all other vertices.
Goal: find shortest path from C to all other vertices.
Shortest-Path

Goal: find shortest path from C to all other vertices.
Shortest-Path

Goal: find shortest path from C to all other vertices.
Goal: find shortest path from C to all other vertices.
As an algorithm:

- Must process all distance $n$ before moving on to distance $n + 1$
As an algorithm:

- Must process all distance $n$ before moving on to distance $n + 1$
- Use queue.
- Must remember distance to each node and whether it has been set yet
Shortest-Path

As an algorithm:

- Must process all distance $n$ before moving on to distance $n + 1$
  Use queue.
- Must remember distance to each node and whether it has been set yet
  Use array.
Shortest-Path: Breadth-First Search

```cpp
vector<int> shortestpaths(int v) {
    vector<int> ret(adj.size(),-1); // -1 => infinity
    ret[v] = 0;
    queue<int> q;
    q.enqueue(v);

    while(!q.empty()) {
        int i = q.dequeue();
        for(int j : adj[i])
            if (ret[j]==-1) {
                ret[j] = ret[i]+1;
                q.enqueue(j);
            }
    }
    return ret;
}
```
Weighted Shortest-Path

Goal: find shortest distance from C to each other vertex
Weighted Shortest-Path

Goal: find shortest distance from C to each other vertex
Weighted Shortest-Path

Goal: find shortest distance from C to each other vertex
Weighted Shortest-Path

Goal: find shortest distance from C to each other vertex
Weighted Shortest-Path

Goal: find shortest distance from C to each other vertex

Graph:

- Vertices: A, B, C, D, E, F
- Edges with weights:
  - A to C: 0.4
  - A to B: 0.8
  - B to C: 2.1
  - B to D: 3.1
  - C to D: 0.2
  - C to E: 0.6
  - C to F: 0.1
  - D to E: 0.2
  - D to F: 0.4
Weighted Shortest-Path

Goal: find shortest distance from C to each other vertex
Weighted Shortest-Path

Goal: find shortest distance from C to each other vertex
Weighted Shortest-Path

Goal: find shortest distance from C to each other vertex
Weighted Shortest-Path

Goal: find shortest distance from C to each other vertex
Goal: find shortest distance from C to each other vertex

(A,0.1), (D,0.2)
Weighted Shortest-Path

Goal: find shortest distance from C to each other vertex

(D,0.2)
Weighted Shortest-Path

Goal: find shortest distance from C to each other vertex

(D,0.2), (D,0.5), (E,0.9)
Weighted Shortest-Path

Goal: find shortest distance from C to each other vertex

(D,0.5), (E,0.9)
Weighted Shortest-Path

Goal: find shortest distance from C to each other vertex

(D,0.5), (E,0.9), (E,0.4), (B,3.3)
Weighted Shortest-Path

Goal: find shortest distance from C to each other vertex

(D, 0.5), (E, 0.9), (B, 3.3)
Weighted Shortest-Path

Goal: find shortest distance from C to each other vertex

(D,0.5), (E,0.9), (B,3.3), (B,0.8)
Weighted Shortest-Path

Goal: find shortest distance from C to each other vertex

(E, 0.9), (B, 3.3), (B, 0.8)
Weighted Shortest-Path

Goal: find shortest distance from C to each other vertex

(E,0.9), (B,3.3)
Weighted Shortest-Path

Goal: find shortest distance from C to each other vertex

(E,0.9), (B,3.3), (C,1.4), (F,2.9), (E,2.9)
Weighted Shortest-Path

Goal: find shortest distance from C to each other vertex

(B,3.3), (F,2.9), (E,2.9)
Weighted Shortest-Path

Goal: find shortest distance from C to each other vertex

(B, 3.3), (E, 2.9)
vector<vector<pair<int,double>>> adj;
// now store target node and weight
vector<double> shortestpaths(int v) {
    vector<double> ret(adj.size(),-1);
    ret[v] = 0.0;
    priority_queue<pair<int,double>> q;
    q.enqueue(make_pair(v,0));

    while(!q.empty()) {
        pair<int,double> s = q.dequeue();
        if (ret[s.first]==-1) {
            ret[s.first] = s.second;
            for(int j : adj[s.first])
                q.enqueue(make_pair(j.first,s.second+j.second));
        }
    }
    return ret;
}