

Outline



- ~~This Week's Problems~~

*(The Uxuhul Voting System, Cudak, Square Fields (Hard),
Funny Games)*

- **Graph Representations**

(adjacency matrix, adjacency list, edge list)

- **Graph Traversal**

(BFS, DFS, Best-First Search)

- **Minimum Spanning Tree**

(Prim, Kruskal)

- **Single-Source Shortest Path**

(Dijkstra, Bellman-Ford)

- **All-Pairs Shortest Path**

(Floyd-Warshall)

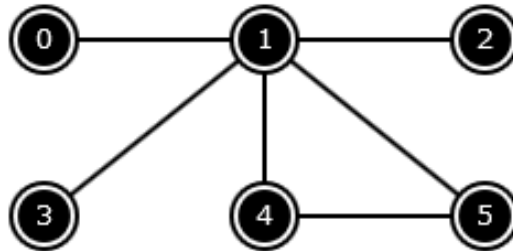
Algorithmic Problem Solving Graphs Part I

Fredrik Präntare & Fredrik Heintz
Department of Computer and Information
Science
Linköping University

Graph Representations



Graph



Adjacency Matrix

	0	1	2	3	4	5
0	0	1	0	0	0	0
1	1	0	1	1	1	1
2	0	1	0	0	0	0
3	0	1	0	0	0	0
4	0	1	0	0	0	1
5	0	1	0	0	1	0

Adjacency List

```
0: 1
1: 0 2 3 4 5
2: 1
3: 1
4: 1 5
5: 1 4
```

Edge List

```
0: 0 1
1: 1 0
2: 1 2
3: 1 3
4: 1 4
5: 1 5
6: 2 1
7: 3 1
8: 4 1
9: 4 5
10: 5 1
11: 5 4
```

Graph Representations



- Adjacency Matrix ($O(V^2)$ space, $O(V)$ to enumerate neighbors)
 - A 2D matrix of weights: `int AdjMat[V][V]`
 - Good for small or dense graphs
 - Cannot represent multi-graphs
- Adjacency List ($O(V+E)$ space, $O(k)$ to enumerate neighbors)
 - A vector of vectors of “node, weight” pairs:
using `AdjList = vector<vector<pair<int, int>>>`
 - Good for large and sparse graphs
 - Can represent multi-graphs
- Edge List ($O(E)$ space, $O(E)$ to enumerate neighbors)
 - A vector of triples “from node, to node, weight”:
using `EdgeList = vector<tuple<int, int, int>>`
 - The vector of edges is usually sorted
 - Useful for some algorithms, like Kruskal’s algorithm for finding MSTs

Graph Traversal



- Depth-First Search ($O(V+E)$ for adjacency lists and $O(V^2)$ for adjacency matrices)
 - Keep a stack of unvisited nodes, initialize it with the start node.
 - Visit the first node in the stack. If it has been visited before then there is a cycle, otherwise add all its children to the stack.
- Breadth-First Search ($O(V+E)$ for adjacency lists and $O(V^2)$ for adjacency matrices)
 - Keep a queue of unvisited nodes, initialize it with the start node.
 - Visit the first node in the queue. If it has been visited before then there is a cycle, otherwise add all its children to the queue.
 - Visits the nodes of a graph in the order of the distance to the start node.
- Best-First Search
 - Keep a priority queue of unvisited nodes, initialize it with the start node.
 - Visit the first node in the queue (the best next node). If it has been visited before then there is a cycle, otherwise add all its children to the queue.
 - Visits the nodes of a graph in the shortest weighted distance from the start node.
- A* Search
 - Like Best-First Search but with a heuristic that underestimates the distance from the current node to the goal node.

Applications of DFS/BFS



- Finding Connected Components – undirected graph
 - Repeatedly select an unvisited node u and run $\text{DFS}(u)$ or $\text{BFS}(u)$ to find all reachable nodes from u . The number of repetitions is the number of components.
- Flood Fill
 - Adaptation of DFS to count the number of cells in a 2D grid with a particular color/property. Usually on implicit graphs, i.e. 2D grids.
- Topological Sorting of Directed Acyclic Graphs
 - Creates a linear ordering of the nodes in a graph such that a node u comes before the node v if there is an edge $u \rightarrow v$.
 - Kahn's algorithm is based on BFS
 - When visiting a node, remove it and all its outgoing edges. Add every node v that now have 0 incoming edges (start with a node that has 0 incoming edges).
 - Tarjan's algorithm also exist, which is based on DFS

Applications of DFS/BFS



- Bipartite Graph Check
 - Try “coloring” the graph with only two colors using BFS, so that no neighbors have the same color.
- Finding Strongly Connected Components in Directed Graphs
 - Tarjan’s SCC algorithm; based on DFS.
 - Solve 2-SAT in linear time [Aspvall, Plass & Tarjan (1979)].

Minimum Spanning Tree



- Given a connected, undirected and weighted graph G , select a subset of edges $E' \subseteq G$ such that the graph G is connected and the total weight of the selected edges E' is minimal.
- This corresponds to finding a *minimal spanning tree* of G , i.e. a tree which connects all the nodes in the graph G and whose total edge weight is minimal.
- Kruskal's Algorithm ($O(E \log V)$)
 - Sort the edges based on non-decreasing weight (use an EdgeList).
 - Greedily add the next edge unless it forms a cycle (use UnionFind)
- Prim's Algorithm ($O(E \log V)$) also exists

Single Source Shortest Path



- Given a weighted graph G and a starting source node s , what are the shortest paths from s to every other node of G ?
- SSSP on Unweighted Graphs (or all edges have equal weight)
 - Use BFS ($O(V+E)$).
 - To reconstruct the shortest path keep a vector $\langle \text{int} \rangle p$ with the parent node of each node and generate the path starting from the destination.
- SSSP on Weighted Graphs
 - Dijkstra ($O((V+E) \log V)$).
 - Maintain a priority queue with reachable nodes sorted on their total distance from the source (increasing).
 - Greedily select the node u with shortest distance d from the source and update the shortest distance to u according to $\text{dist}[u] = \min(\text{dist}[u], d)$.
 - If $\text{dist}[u]$ is decreased add all neighboring nodes to the priority queue.

Single Source Shortest Path



- SSSP on Weighted Graphs with Negative Cycles
 - The Dijkstra version described above works even with negative weights, but not with negative cycles.
 - Bellman-Ford ($O(VE)$ for adjacency lists)
 - Idea: Relax all E edges $V-1$ times.
 - Basically do a DP over every edge (u, v) $V-1$ times and update as follows:
 - $\text{dist}[v] = \min(\text{dist}[v], \text{dist}[u] + w[u][v])$
 - It is possible to determine if there are negative cycles since if there are no negative cycles then after $V-1$ iterations no relaxations should be possible. This means that if on the V th iteration relaxations are possible, there is a negative cycle.

All-Pairs Shortest Path



- Given a weighted graph G , find the shortest path between every pair of nodes u and v .
- Floyd-Warshall ($O(V^3)$)
 - For every triple k, i, j compute
$$\text{AdjMat}[i][j] = \min(\text{AdjMat}[i][j], \text{AdjMat}[i][k] + \text{AdjMat}[k][j])$$
- Applications of APSP
 - Solving the SSSP on small weighted graphs
 - Transitive closure
 - Minimax and maximin
$$\text{AdjMat}[i][j] = \min(\text{AdjMat}[i][j], \max(\text{AdjMat}[i][k], \text{AdjMat}[k][j]))$$
 - Finding the cheapest/negative cycle
 - Finding the diameter of a graph
 - Finding the strongly connected components of a graph

Summary SSSP/APSP



Graph Criteria	BFS $O(V+E)$	Dijkstra $O((V+E) \log V)$	Bellman-Ford $O(VE)$	Floyd Warshall $O(V^3)$
Max size	$V, E \leq 10M$	$V, E \leq 300K$	$VE \leq 10M$	$V \leq 400$
Unweighted	Best	Ok	Bad	Bad in general
Weighted	WA	Best	Ok	Bad in general
Negative weight	WA	Our variant ok	Ok	Bad in general
Negative cycle	Cannot detect	Cannot detect	Can detect	Can detect
Small graph	WA if weighted	Overkill	Overkill	Best

Summary



- Graph Representations (adjacency matrix, adjacency list, edge list)
- Graph Traversal (BFS, DFS, Best First Search)
- Finding Strongly Connected Components
- Minimum Spanning Tree (Prim, Kruskal; lab 2.5)
- Single-Source Shortest Path (Dijkstra, Bellman-Ford; lab 2.1-2.3)
- All-Pairs Shortest Path (Floyd-Warshall, lab 2.4)