COSC-302

EXAM 2 REVIEW

by Tyler Senter

Overview

Graphs
- Undirected vs. Directed
- Weighted vs. Unweighted
- Cyclic vs. Acyclic
- Dense vs. Sparse
  - Implementation

Disjoint Sets

Algorithms
- Breadth-First vs. Depth-First Searches
  - Implementation
- Dijkstra's Shortest Path
- Topological Sorting

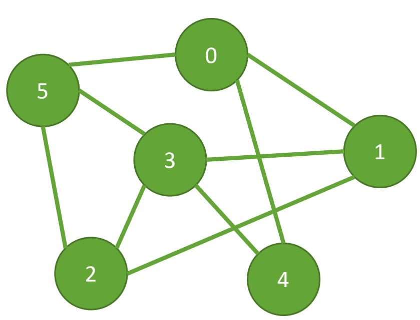
Overview (cont.)

Algorithms (cont.)
- Minimum Spanning Trees
  - Prim
  - Kruskal
- Maximum Flow: Edmond-Karp's Breadth-First Search
  - Min-cut
- Floyd-Warshall's Shortest Path
- Computational Costs

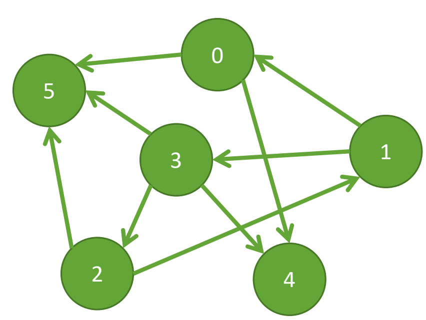
Dynamic Programming

Graphs – Undirected vs. Directed

Undirected
- A vertex can be visited by any of its adjacent vertices

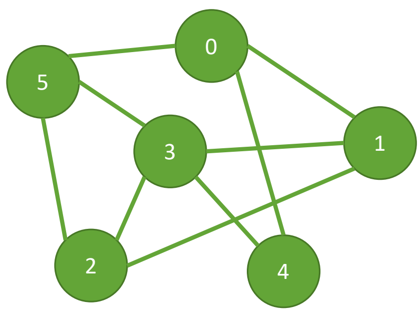


Directed
- A vertex can only visit vertices with a **direct** connection

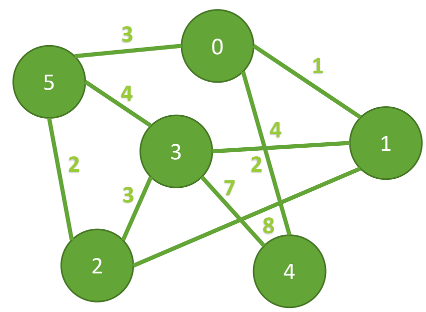


Graphs – Unweighted vs. Weighted

Unweighted
- There is no difference in cost for visiting different vertices



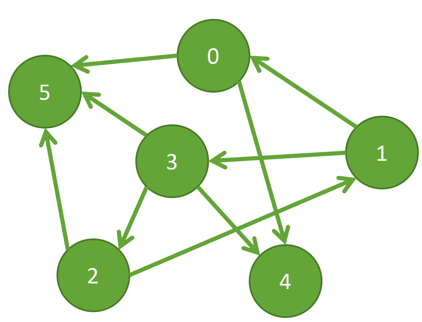
Weighted
- A **cost** exists between vertices; represents time, distance, money, etc.



Graphs – Cyclic vs. Acyclic

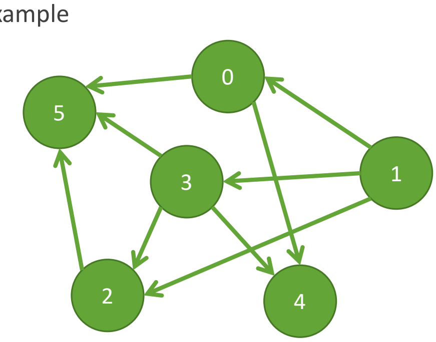
Cyclic Graphs
- Contain at least one **cycle**, meaning a DFS may reach an *ancestor* (vertex colored black)

Example



Acyclic Graph
- Contain **no** cycles

Example



Graphs – Dense vs. Sparse

Dense Graphs

Definition
- The number of edges for each vertex is close to the maximal number of edges
- $V \ll E < V^2$

Implementation
- Adjacency matrix – mapping between every pair of vertices marking whether or not an edge exists

Sparse Graphs

Definition
- The number of edges for each vertex is close to the minimal number of edges
- $V < E \ll V^2$

Implementation
- Adjacency lists – array containing all vertices each vertex is adjacent to

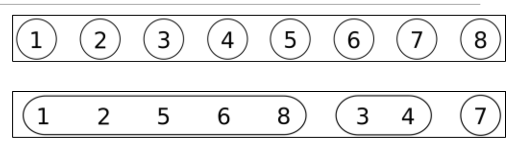
Example: Facebook
- Facebook has an estimated 2.5 billion accounts, with an average of 338 friends per account.

Disjoint Sets

Disjoint Sets track different, non-overlapping elements in a “universe”

Useful for generating mazes (Lab 6) by starting with each cell as its own set and merging (**union**) until every cell is in the same set (**find**)

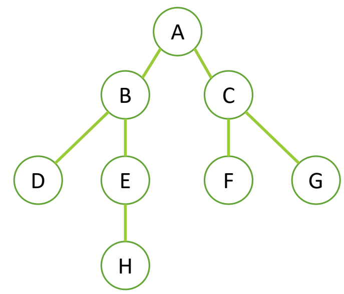
Operations
- **Union/merge** – take two unconnected sets and combine them
- **Find** – Locate the set containing a particular value



Algorithms – Breadth-First Search

Algorithm

1. Let Q be a queue
2. Add source to Q
3. While Q is not empty
    1. Let v be front of Q, remove front of Q
    2. Process v
    3. If v has not been visited
        1. Mark v as visited
        2. For every vertex j adjacent to v:
            1. Push j onto Q
        3. Go to 3

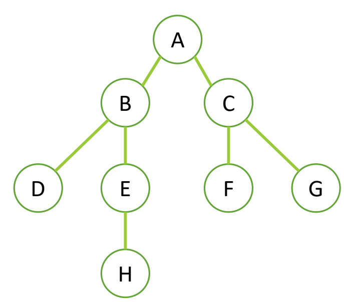


A	B	C	D	E	F	G	H

Algorithms – Depth-First Search

Algorithm

1. Let S be a stack
2. Add source to S
3. While S is not empty
    1. Let v be top of S, remove front of S
    2. Process v
    3. If v has not been visited
        1. Mark v as visited
        2. For every vertex j adjacent to v:
            1. Push j onto S
        3. Go to 3



A	B	D	E	H	C	F	G

```
void bfs(int source) {
    queue<int> Q;
    vector<bool> visited;
    Q.push(source);

    visited.assign(V.size(), false);

    while (!Q.empty()) {
        int i = Q.front();
        Q.pop();

        if (!visited[i]) {
            visited[i] = true;
            for (int k = 0; k < E[i].size(); k++)
                Q.push(E[i][k]);
        }
    }
}
```

```
void dfs(int source) {
    stack<int> S;
    vector<bool> visited;
    S.push(source);

    visited.assign(V.size(), false);

    while (!S.empty()) {
        int i = S.top();
        S.pop();

        if (!visited[i]) {
            visited[i] = true;
            for (int k = 0; k < E[i].size(); k++)
                S.push(E[i][k]);
        }
    }
}
```

Algorithms – BFS and DFS Traversals

Algorithms - Dijkstra’s Shortest Path

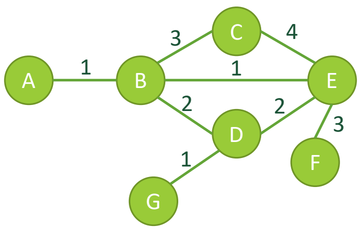
Guaranteed to find the shortest (least weighted) path between two vertices
- Works for directed and undirected graphs
- Does **not** work with negative weights

Algorithm
- Color all vertices white
- Set $\text{distance}(source) = 0, \text{distance}(\text{other}) = \infty$

1. Find white vertex v with the lowest distance
2. For each white vertex adjacent to w,
    1. Calculate distance: $d_{new} = \text{distance}(v) + \text{weight}(v, w)$
    2. If $d_{new} < \text{distance}(w)$,
        1. Set $\text{distance}(w) = d_{new}$
3. Color v black
4. Repeat to 1

Algorithm ends when no white vertices exist

Algorithms – Dijkstra's SP (Undirected)



Source = A

	A	B	C	D	E	F	G
init	0	∞	∞	∞	∞	∞	∞
0		1	∞	∞	∞	∞	∞
1			4	3	2	∞	∞
2			4	3		5	∞
3			4			5	4
4						5	4
5						5	

The source vertex is always a distance of 0

All other vertices start with distance ∞ since we don't know a path

Vertex *B* has the shortest distance, so we look at the adjacent vertices

Vertex *E* has the shortest total distance

Vertex *D* has the shortest total distance

Vertex *C* has the shortest total distance

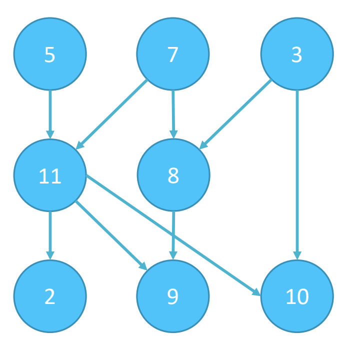
Vertex *G* has the shortest total distance

Algorithms – Topological Sort

Basic Idea

- Sort vertices based off of **indegree** – number of "dependencies"
- Can find the **critical path** for a directed, acyclic graph

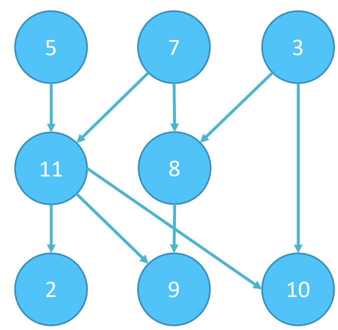
2	3	5	7	8	9	10	11
1	0	0	0	2	2	2	2
1		0	0	1	2	1	2
1			0	1	2	1	1
1				0	2	1	0
1					1	1	0
0					0	0	
					0	0	
						0	
3	**5**	**7**	**8**	**11**	**2**	**9**	**10**



Algorithms – Topological Sort (cont.)

Pseudocode:

1. Add vertices to indegree 0 to queue Q
2. While Q is not empty
    1. Let vertex v equal to the front of Q
        1. For each vertex w adjacent to v:
            1. Decrement indegree of w
            2. If $\text{indegree}(w) == 0$
                1. Add w to back of Q
            3. If $\text{dist}(v) + \text{weight}(w, v) > \text{dist}(w)$
                1. $\text{dist}(w) = \text{dist}(v) + \text{weight}(w, v)$
                2. $\text{vlink}(w) = v$



Algorithms – Prim (MST)

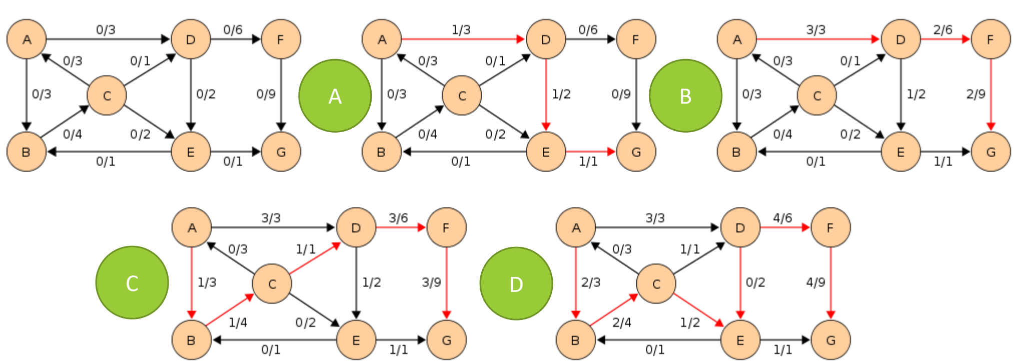
Pseudocode:

1. Color all vertices white
2. $\text{dist}(\text{source}) = 0, \text{dist}(\text{others}) = \infty$
3. While non-white vertices exist:
    1. Select v as min. distance white vertex
    2. Color v black
    3. For every white vertex w adjacent to v
        1. if $\text{dist}(w) > \text{weight}(v, w)$
            1. $\text{dist}(w) = \text{weight}(v, w)$
        2. Return to 3.3
    4. Return to 3

Algorithms – Kruskal (MST)

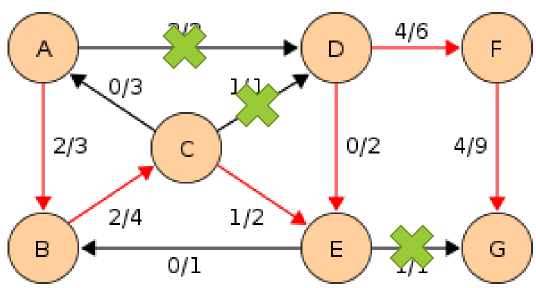
Algorithm
- Make each vertex its own set inside a disjoint set
- Sort edges by weight
- For each edge (v, w) by order of weight:
  - If $\text{find}(v) \neq \text{find}(w)$:
    - $\text{union}(v, w)$

Algorithms – Edmonds-Karp (MF)



Maximum Flow $= 5$

Algorithms – Edmonds-Karp (MC)



Algorithm
- Starting at the source, go through the graph until you reach a full edge (where flow = capacity)
- "Mark off" these edges

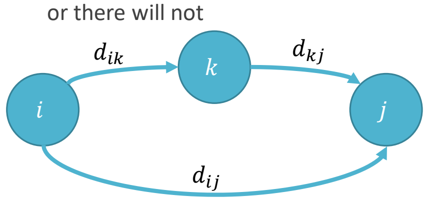
Calculation
- To calculate the minimum cut, add the flows of all of these "marked off" edges
- If the Max Flow equals the Min Cut, you have found the true Max Flow

$$\text{Min Cut} = f(A,D) + f(C,D) + f(E,G)$$
$$= \quad 3 \quad + \quad 1 \quad + \quad 1$$

Algorithms – Floyd-Warshall (SP)

Purpose
- Computes shortest path between every pair of vertices

Basic Idea
- Pick two vertices i and j
- There will either exist a path k between i and j, or there will not



Pseudocode
1. $D = [w_{ij}]$
2. For every positive integer k where $k < N$
   1. For every positive integer i where $i < N$
      1. For every positive integer j where $j < N$
         1. If $D[i][j] > D[i][k] + D[k][j]$
            1. $D[i][j] = D[i][k] + D[k][j]$
            2. $L(i,j) = L(k,j)$

$$L(i,j) = \begin{cases} -1 & i = j \text{ or } w_{ij} = \infty \\ i & \text{otherwise} \end{cases}$$

Computational Costs

Name	Type	Cost
Disjoint Sets	Add Set	$O(n)$
	Union/Find	$O\big(\alpha(n)\big)\sim O(1)$
BFS	Search	$O(V+E)$
DFS	Search	$O(V+E)$
Dijkstra	SP	$O(V^2)$
Topological Sort	Sort	$O(V+E)$
Prim	MST	$O(V^2)$
Kruskal	MST	$O(E\log V)$
Edmonds-Karp	MF	$O(VE^2)$
Floyd-Warshall	SP	$O(V^3)$

* SP Shortest Path
MST Minimum Spanning Tree
MF Maximum Flow

Dynamic Programming

Used to make code more efficient, both in time complexity and memory usage/lookup

Steps:

1. Identify recursive solution with overlapping subproblems
2. Use memoization (cache results for reuse)
3. Use iteration instead of recursion
4. Reduce cache size
5. Return to recursion
    1. Less efficient, but makes code more elegant

Primary 3

Those who cannot remember the past are condemned to repeat it.

-Dynamic Programming