CSE 421
Algorithms

Richard Anderson

Autumn 2016

Lecture 6

Announcements

- Reading
  - Start on Chapter 4

Stable Matching Results

- Averages of 5 runs
- Much better for M than W
- Why is it better for M?

- What is the growth of m-rank and w-rank as a function of n?

n	m-rank	w-rank
500	5.102	98.048
500	7.52	66.952
500	8.57	58.176
500	6.322	75.874
500	5.25	90.726
500	6.5528	77.9552
1000	6.796	146.936
1000	6.502	154.714
1000	7.14	133.538
1000	7.444	128.961
1000	7.364	137.852
1000	7.0492	140.4002
2000	7.826	257.7955
2000	7.505	263.781
2000	11.4245	175.1735
2000	7.1665	274.7615
2000	7.547	261.602
2000	8.2938	246.6227

Graph Theory

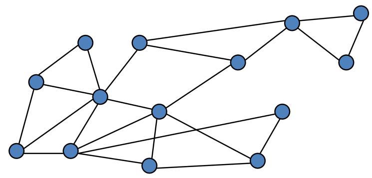
- G = (V, E)
  - V: vertices, |V|= n
  - E: edges, |E| = m
- Undirected graphs
  - Edges sets of two vertices {u, v}
- Directed graphs
  - Edges ordered pairs (u, v)
- Many other flavors
  - Edge / vertices weights
  - Parallel edges
  - Self loops
- Path: $v_1, v_2, \ldots, v_k$, with (v_i, v_{i+1}) in E
  - Simple Path
  - Cycle
  - Simple Cycle
- Neighborhood
  - N(v)
- Distance
- Connectivity
  - Undirected
  - Directed (strong connectivity)
- Trees
  - Rooted
  - Unrooted

Last Lecture

- Bipartite Graphs : two-colorable graphs
- Breadth First Search algorithm for testing two-colorability
  - Two-colorable iff no odd length cycle
  - BFS has cross edge iff graph has odd cycle

Graph Search

- Data structure for next vertex to visit determines search order



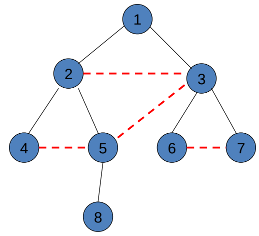
Graph search

```
Breadth First Search
    S = {s}
    while S is not empty
        u = Dequeue(S)
        if u is unvisited
            visit u
            foreach v in N(u)
                Enqueue(S, v)
```

```
Depth First Search
    S = {s}
    while S is not empty
        u = Pop(S)
        if u is unvisited
            visit u
            foreach v in N(u)
                Push(S, v)
```

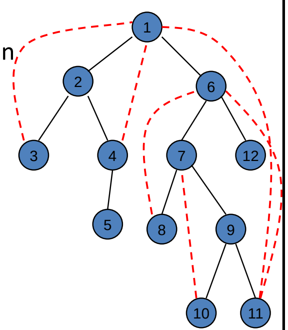
Breadth First Search

- All edges go between vertices on the same layer or adjacent layers



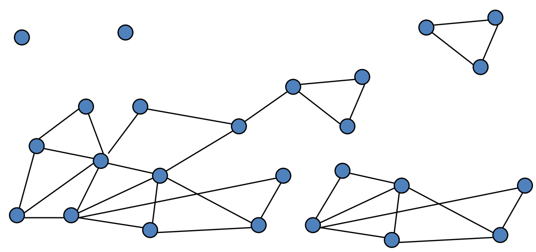
Depth First Search

- Each edge goes between vertices on the same branch
- No cross edges



Connected Components

- Undirected Graphs

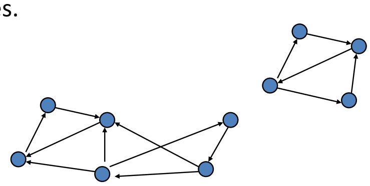


Computing Connected Components in O(n+m) time

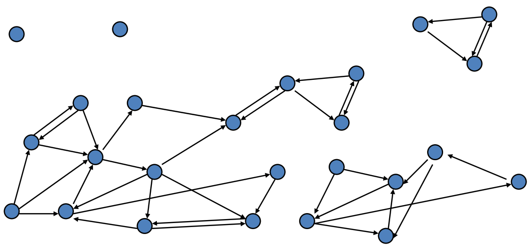
- A search algorithm from a vertex v can find all vertices in v's component
- While there is an unvisited vertex v, search from v to find a new component

Directed Graphs

- A Strongly Connected Component is a subset of the vertices with paths between every pair of vertices.



Identify the Strongly Connected Components

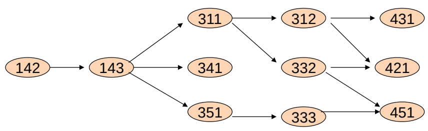


Strongly connected components can be found in O(n+m) time

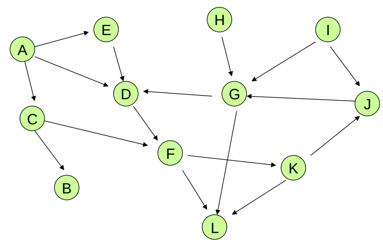
- But it's tricky!
- Simpler problem: given a vertex v, compute the vertices in v's scc in O(n+m) time

Topological Sort

- Given a set of tasks with precedence constraints, find a linear order of the tasks

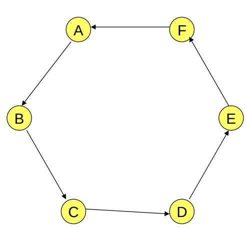


Find a topological order for the following graph



If a graph has a cycle, there is no topological sort

- Consider the first vertex on the cycle in the topological sort
- It must have an incoming edge



Lemma: If a graph is acyclic, it has a vertex with in degree 0

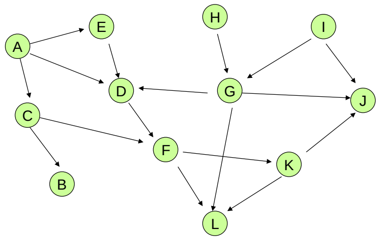
- Proof:
  - Pick a vertex v_1, if it has in-degree 0 then done
  - If not, let (v_2, v_1) be an edge, if v_2 has in-degree 0 then done
  - If not, let (v_3, v_2) be an edge . . .
  - If this process continues for more than n steps, we have a repeated vertex, so we have a cycle

Topological Sort Algorithm

While there exists a vertex v with in-degree 0

Output vertex v

Delete the vertex v and all out going edges



Details for O(n+m) implementation

- Maintain a list of vertices of in-degree 0
- Each vertex keeps track of its in-degree
- Update in-degrees and list when edges are removed
- m edge removals at O(1) cost each