CSE 421
Introduction to Algorithms

Winter 2024

Lecture 3

Announcements

- Reading
  - Chapter 3 (Mostly review)
  - Start on Chapter 4
- Office Hours:

Richard Anderson	CSE2 344, Mon 3:30-4:30	CSE2 344, Fri 2:30-3:30
Raymond Gao	Allen 3rd Floor, Tue 5:30-6:30	CSE2 150, Thu 5:30-6:30
Sophie Robertson	Allen 4th Floor, Mon 11:30-1:30	
Aman Thukral	Allen 2nd Floor, Fri 3:30-5:30	
Kaiyuan Liu	Allen 2nd Floor, Tues 9:30-11:30	
Tom Zhaoyang Tian	CSE2 153, Wed 9:30-11:30	
Albert Weng	CSE2 131, Mon 10:30-11:30	CSE2 131, Fri 10:30-11:30

Schedule

- Monday
  - Run time/Big-Oh (most of this deferred to section)
  - Graph theory
  - Search/Bipartite Matching
- Wednesday
  - Connectivity
  - Topological Sort
- Friday
  - Greedy Algorithms

Run time / Big Oh

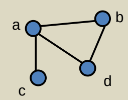
- Run time function T(n)
  - T(n) is the maximum time to solve an instance of size n
- Disregard constant functions
- T(n) is O(f(n)) $[T : Z^+ \rightarrow R^+]$
  - If n is sufficiently large, T(n) is bounded by a constant multiple of f(n)
  - Exist c, n_0, such that for $n > n_0$, $T(n) < c\, f(n)$
- T(n) is Ω(f(n)) $[T : Z^+ \rightarrow R^+]$
  - If n is sufficiently large, T(n) is at least a constant multiple of f(n)
  - Exist $\epsilon > 0$, n_0, such that for $n > n_0$, $T(n) > \epsilon\, f(n)$

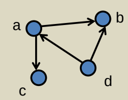
Graph Theory

- G = (V, E)
  - V – vertices
  - E – edges
- 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

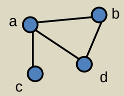
Definitions

- Path: $v_1, v_2, \ldots, v_k$, with (v_i, v_{i+1}) in E
  - Simple Path
  - Cycle
  - Simple Cycle
- Neighborhood
  - N(v)
  - $N^+(v)$, $N^-(v)$
- Distance
- Connectivity
  - Undirected
  - Directed (strong connectivity)
- Trees
  - Rooted
  - Unrooted



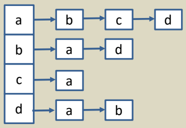


Graph Representation



V = { a, b, c, d}

E = { {a, b}, {a, c}, {a, d}, {b, d} }



Adjacency List

	1	1	1
1		0	1
1	0		0
1	1	0	

Incidence Matrix

7

Graph search

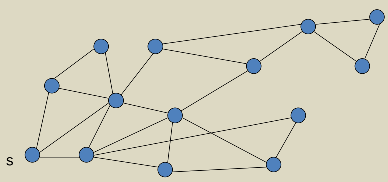
- Find a path from s to t

```
S = {s}
while S is not empty
        u = Select(S)
        visit u
        foreach v in N(u)
                if v is unvisited
                        Add(S, v)
                        Pred[v] = u
                if (v = t) then path found
```

8

Breadth first search

- Explore vertices in layers
  - s in layer 1
  - Neighbors of s in layer 2
  - Neighbors of layer 2 in layer 3 . . .



9

Breadth First Search

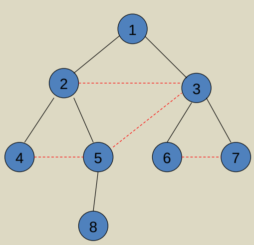
- Build a BFS tree from s

```
Initialize Level[v] = -1 for all v;
Q = {s}
Level[s] = 1;
while Q is not empty
        u = Q.Dequeue()
        foreach v in N(u)
                if (Level[v] == -1)
                        Q.Enqueue(v)
                        Pred[v] = u
                        Level[v] = Level[u] + 1
```

10

Key observation

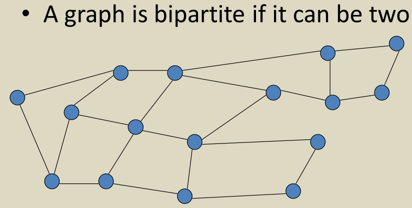
- All edges go between vertices on the same layer or adjacent layers



11

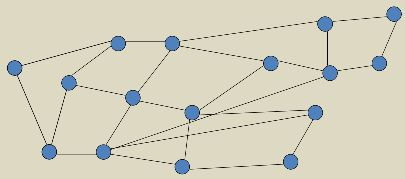
Bipartite Graphs

- A graph V is bipartite if V can be partitioned into V_1, V_2 such that all edges go between V_1 and V_2
- A graph is bipartite if it can be two colored



12

Can this graph be two colored?



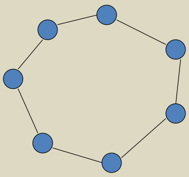
Algorithm

- Run BFS
- Color odd layers red, even layers blue
- If no edges between the same layer, the graph is bipartite
- If edge between two vertices of the same layer, then there is an odd cycle, and the graph is not bipartite

Theorem: A graph is bipartite if and only if it has no odd cycles

Lemma 1

- If a graph contains an odd cycle, it is not bipartite



Lemma 2

- If a BFS tree has an *intra-level edge*, then the graph has an odd length cycle

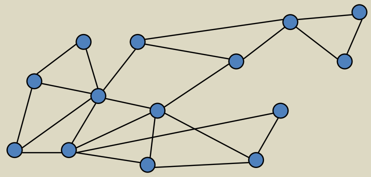
Intra-level edge: both end points are in the same level

Lemma 3

- If a graph has no odd length cycles, then it is bipartite

Graph Search

- Data structure for next vertex to visit determines search order



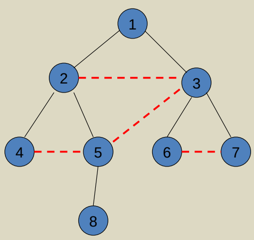
Graph search

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