CSE 332: Data Structures and Parallelism

Fall 2022
Richard Anderson
Lecture 25: Graph Traversal and Shortest Paths and Other Algorithms

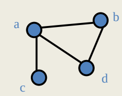
Announcements

- Lectures
  - ~~Intro to graphs~~
  - ~~Topological Sort~~
  - ~~Parallelism and Concurrency (6 lectures)~~
  - Graph Algorithms
    - Graph Traversal
    - Shortest Paths
    - Minimum Spanning Tree
  - Theory of NP-Completeness (2 lectures)

Graph Theory

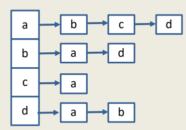
- G = (V, E)
  - V: vertices, |V| = n
  - E: edges, |E| = m
- Undirected graphs
  - Edges sets of two vertices {u, v}
- Directed graphs
  - Edges ordered pairs (u, v)
- Many other flavors
  - Edge / vertices weights
  - Parallel edges
  - Self loops
- Path: $v_1, v_2, \ldots, v_k$, with $(v_i, v_{i+1})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

Graph Representation



V = { a, b, c, d}

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



Adjacency List

O(n + m) space

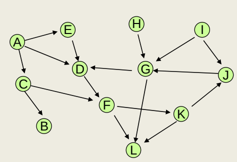
	1	1	1
1		0	1
1	0		0
1	1	0	

Adjacency Matrix

O(n^2) space

Topological Sort

```
while there is a vertex v with in-degree 0 {
    output v
    remove v from G
}
```

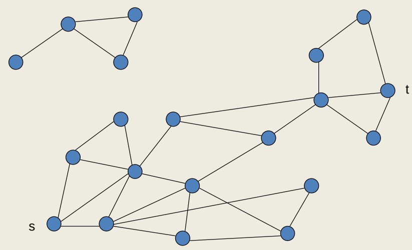


Graph search

- Find a path from s to t

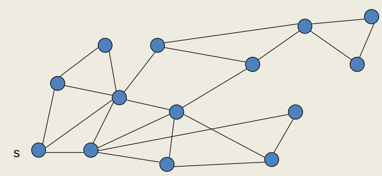
```
S = {s}
while S is not empty
        u = Select(S)
        visit u
        foreach v in N(u)
                if v is unvisited
                        Add(S, v)
                        Pred[v] = u
                if (v == t) then path found
```

Graph Search



Breadth first search

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



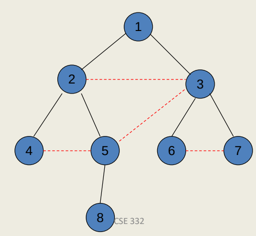
Breadth First Search

- Build a BFS tree from s

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

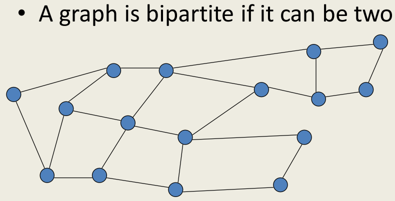

Key observation: BFS Levels

- All edges go between vertices on the same level or adjacent levels

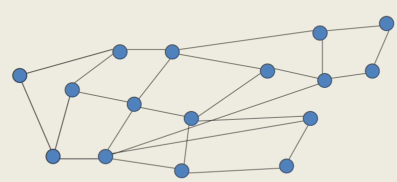


Bipartite Graphs

- 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



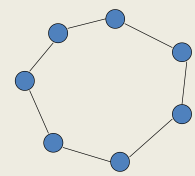
Can this graph be two colored?



Algorithm

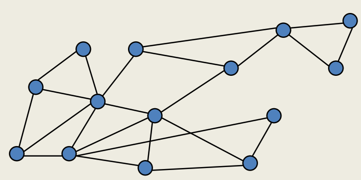
- Run BFS
- Color odd layers red, even layers blue
- If no edges between the same layer, the graph is bipartite
- If edge between two vertices of the same layer, then there is an odd cycle, and the graph is not bipartite

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



Graph Search

- Data structure for next vertex to visit determines search order



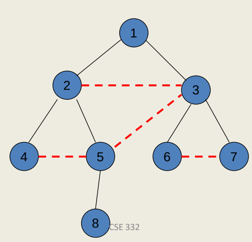
Graph search

```
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)
```

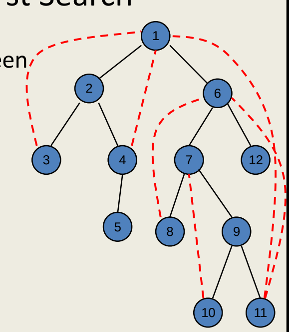

Breadth First Search

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



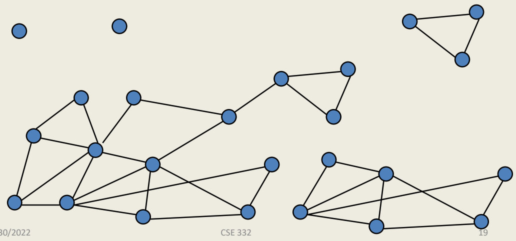
Depth First Search

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



Connected Components

- Undirected Graphs

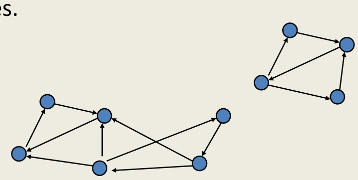


Computing Connected Components in O(n+m) time

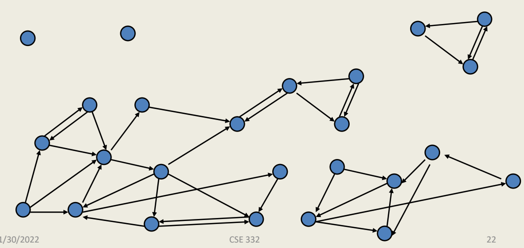
- A search algorithm from a vertex v can find all vertices in v's component
- While there is an unvisited vertex v, search from v to find a new component

Directed Graphs

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



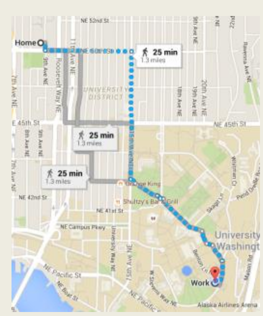
Identify the Strongly Connected Components



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

Paths and connectivity



The Shortest Path Problem

Given a graph G, and vertices s and t in G, find the shortest path from s to t.

Two cases: weighted and unweighted.

For a path $p = v_0\ v_1\ v_2 \ldots v_k$

- *unweighted length* of path $p = k$ (*length*)
- *weighted length* of path $p = \sum_{i=0..k-1} c_{i,i+1}$ (*cost*)

Single Source Shortest Paths (SSSP)

Given a graph G and vertex s, find the shortest paths from s to all vertices in G.

- How much harder is this than finding single shortest path from s to t?

Variations of SSSP

- Weighted vs unweighted
- Directed vs undirected
- Cyclic vs acyclic
- Positive weights only vs negative weights allowed
- Shortest path vs longest path
- …

Applications

- Network routing
- Driving directions
- Cheap flight tickets
- Critical paths in project management (see textbook)
- …

SSSP: Unweighted Version

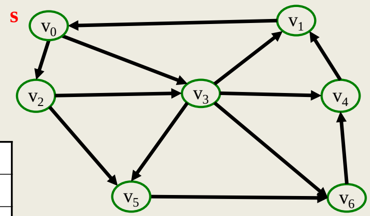

```
void UnweightedGraphSearch(Vertex s){
  Queue q(NUM_VERTICES);
  Vertex v, w;
  q.enqueue(s);
  s.dist = 0;

  while (!q.isEmpty()){
    v = q.dequeue();
    for each w adjacent to v
      if (w.dist == INFINITY){
        w.dist = v.dist + 1;
        w.prev = v;
        q.enqueue(w);
      }
  }
}
```

each edge examined at most once – if adjacency lists are used

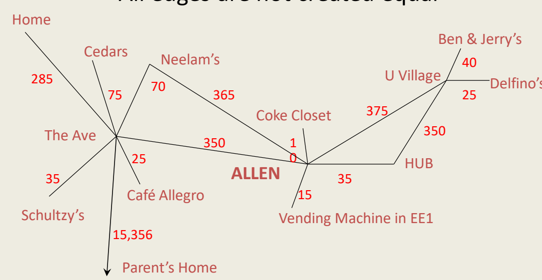
each vertex enqueued at most once

total running time: O( )



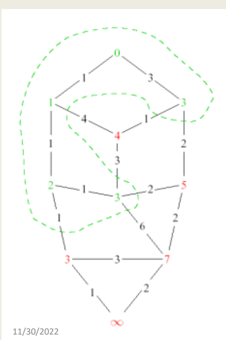
V	Dist	prev
v_0		
v_1		
v_2		
v_3		
v_4		
v_5		
v_6		

Weighted SSSP:
All edges are not created equal



Can we calculate shortest distance to all vertices from Allen Center?

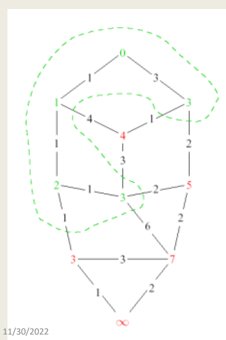
Dijkstra's Algorithm: Idea



Adapt BFS to handle weighted graphs

Two kinds of vertices:
- Known
  - shortest distance is already known
- Unknown
  - Have tentative distance

Dijkstra's Algorithm: Idea



At each step:
1) Pick closest unknown vertex
2) Add it to known vertices
3) Update distances

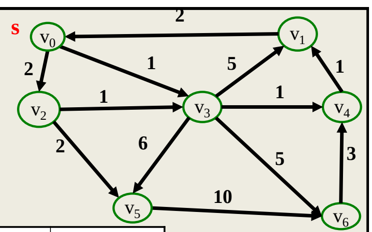
Dijkstra's Algorithm: Pseudocode

```
Initialize the cost of each node to ∞
Initialize the cost of the source to 0

While there are unknown vertices left in the graph
    Select an unknown vertex a with the lowest cost
    Mark a as known
    For each vertex b adjacent to a
        newcost = cost(a) + cost(a,b)
        if (newcost < cost(b))
                cost(b) = newcost
                previous(b) = a
```


Important Features

- Once a vertex is known, the cost of the shortest path to that vertex is known
- While a vertex is still unknown, another shorter path to it might still be found
- The shortest path can found by following the previous pointers stored at each vertex



V	Known?	Cost	Previous
v0			
v1			
v2			
v3			
v4			
v5			
v6			

Dijkstra's Alg: Implementation

Initialize the cost of each vertex to ∞
Initialize the cost of the source to 0
While there are unknown vertices left in the graph
    Select the unknown vertex ***a*** with the lowest cost
    Mark ***a*** as known
    For each vertex ***b*** adjacent to ***a***
        newcost = min(cost(***b***), cost(***a***) + cost(***a***, ***b***))
        if newcost < cost(**b**)
            cost(***b***) = newcost
            previous(***b***) = *a*

What data structures should we use?

Running time?

Dijkstra's Algorithm: Summary

- Classic algorithm for solving SSSP in weighted graphs *without negative weights*
- A *greedy* algorithm (irrevocably makes decisions without considering future consequences)
- Why does it work?

Continuation

- I don't expect to get close to this on Wednesday
  - I do not plan on giving the correctness proof – you will need to wait for 421. I might wave my hands a bit on the general ideas for the proof
  - Assuming I have time on Friday, I am going to talk more about the use of heaps in Dijkstra's algorithm, as this is a data structures course