CS 161 Section 5

CA: [name of CA]
Lecture Recap
Undirected Graphs

• Have *vertices* and *edges*
  • $V$ is the set of vertices
  • $E$ is the set of edges
  • Formally, a graph is $G = (V,E)$

• Example
  • $V = \{1,2,3,4\}$
  • $E = \{\{1,3\}, \{2,4\}, \{3,4\}, \{2,3\}\}$

• The *degree* of vertex 4 is 2.
  • There are 2 edges coming out.
  • Vertex 4’s *neighbors* are 2 and 3
Directed Graphs

• Have **vertices and edges**
  • *V* is the set of vertices
  • *E* is the set of **DIRECTED** edges
  • Formally, a graph is *G* = (*V*,*E*)

• Example
  • *V* = {1,2,3,4}
  • *E* = { (1,3), (2,4), (3,4), (4,3), (3,2) }

• The **in-degree** of vertex 4 is 2.
• The **out-degree** of vertex 4 is 1.
• Vertex 4’s **incoming neighbors** are 2,3
• Vertex 4’s **outgoing neighbor** is 3.
## Trade-offs

Say there are $n$ vertices and $m$ edges.

<table>
<thead>
<tr>
<th></th>
<th>( n = # \text{ of vertices} )</th>
<th>( m = # \text{ of edges} )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Edge membership</strong></td>
<td><img src="image" alt="Matrix" /></td>
<td></td>
</tr>
<tr>
<td>Is ( e = {v, w} ) in ( E )?</td>
<td>( O(1) )</td>
<td>( O(\text{deg}(v)) ) or ( O(\text{deg}(w)) )</td>
</tr>
<tr>
<td><strong>Neighbor query</strong></td>
<td><img src="image" alt="Graph" /></td>
<td></td>
</tr>
<tr>
<td>Give me ( v )'s neighbors.</td>
<td>( O(n) )</td>
<td>( O(\text{deg}(v)) )</td>
</tr>
<tr>
<td><strong>Space requirements</strong></td>
<td>( O(n^2) )</td>
<td>( O(n + m) )</td>
</tr>
</tbody>
</table>

Generally better for *sparse* graphs.

We’ll assume this representation for the rest of the class.
Depth-First Search: Pseudocode with start and end times

- **DFS**(w, currentTime):
  - w.startTime = currentTime
  - currentTime ++
  - Mark w as *in progress*.
  - for v in w.neighbors:
    - if v is *unvisited*:
      - currentTime = DFS(v, currentTime)
      - currentTime ++
  - w.finishTime = currentTime
  - Mark w as *all done*
  - return currentTime
Example DFS (we sort edges in alphabetical order)

Start: 1
Example DFS (we sort edges in alphabetical order)

Start:
1

A

B

C

D

E

F
Example DFS (we sort edges in alphabetical order)
Example DFS (we sort edges in alphabetical order)

Start: 1
A

Start: 3
B

Start: 4
C

Start: 2
D

E

F
Example DFS (we sort edges in alphabetical order)

Start:
- A: 1
- B: 3
- D: 2
- E
- F

Finish:
- C: 4, 5
Example DFS (we sort edges in alphabetical order)

Start: 1
A

Start: 3
B

Finish: 5
C

Start: 4

Start: 2
D

Start: 6
E

F
Example DFS (we sort edges in alphabetical order)
Example DFS (we sort edges in alphabetical order)
Example DFS (we sort edges in alphabetical order)

- **Start**: 1, **Finish**: 12
- **Start**: 2, **Finish**: 11
- **Start**: 3, **Finish**: 10
- **Start**: 4, **Finish**: 5
- **Start**: 6, **Finish**: 9
- **Start**: 7, **Finish**: 8
Relation in DFS tree and start/finish time

- u is v’s ancestor if $\text{start}(u) < \text{start}(v) < \text{finish}(v) < \text{finish}(u)$
- u is v’s descendant if $\text{start}(v) < \text{start}(u) < \text{finish}(u) < \text{finish}(v)$
- u is v’s cousin if $\text{start}(v) < \text{finish}(v) < \text{start}(u) < \text{finish}(u)$
- Notice that startTime and finishTime never interleave $(\text{start}(v)<\text{start}(u)<\text{finish}(v)<\text{finish}(u))$ will never happen.
Breadth-First Search: Pseudocode

- **BFS(G, v)**
- **Input:** graph G = (V, E); node v in V, edge e in E
- **Output:**
  - Array visited: V -> {True, False}
  - Layer i of vertices. Initialize $L_0$ with start vertex v.
Breadth-First Search: Pseudocode

- BFS(G, v)
- Input: graph G = (V, E); node v in V, edge e in E
- Output:
  - Array visited: V -> {True, False}
  - Layer of vertices to visit.
    - Initialize $L_0$ with start vertex v. Mark v as visited
  - While $L_i$ is non-empty: # keep looping as we have unexplored vertices
    - Visit neighbours of elements in $L_i$ # vertices currently being explored
Breadth-First Search: Pseudocode

- BFS(G, v)
- Input: graph G = (V, E); node v in V, edge e in E
- Output:
  - Array visited: V -> {True, False}
  - Layer of vertices to visit.
    - Initialize L₀ with start vertex v. Mark v as visited
    - While Lᵢ is non-empty:  # keep looping as we have unexplored vertices
      - for u in Lᵢ:  # vertices currently being explored
        - for v neighbour of u:
          - if not visited v:  # only work on vertices we have not explored
            - mark v visited and add to Lᵢ₊₁
        - i += 1

Can we use queues instead of lists?
Example BFS (we sort edges in alphabetical order)
Example BFS (we sort edges in alphabetical order)
Example BFS (we sort edges in alphabetical order)
Example BFS (we sort edges in alphabetical order)
What can we do with BFS?

• Find all vertices reachable from the start vertex
• Find the shortest paths from a vertex to all the others
• Check if a graph is bipartite graph.
  • Color the levels of the BFS tree in alternating colors.
  • If you never color two connected nodes the same color, then it is bipartite.
  • Otherwise, it’s not.
What can we do with DFS?

- Find all vertices reachable from the start vertex
- Topological sort for directed graph without any cycle (DAG)
  - In reverse order of finishing time in DFS!
- Both algorithms run in $O(|V| + |E|)$. 
Strongly Connected Components
Strongly vs Connected Components

**Undirected graphs**
Connected Components (CC)

**Directed graphs**
Strongly Connected Components (SCC)

Find with BFS/DFS

Find with DFS (today)

*Please do not confuse them!*
Algorithm

Running time: $O(n + m)$

- Run DFS.
  - Choose starting vertices in any order.
  - Order vertices by reverse finishing times.
- Reverse all the edges in the graph.
- Do DFS again to create a new DFS forest.
  - (Using the reverse-finish-time order from the first DFS run.)
- The SCCs are the different trees in the new DFS forest.

(This is basically “PretendTopoSort”)

(This doesn’t change SCCs)
Example
Example
Example

1. Start with an arbitrary vertex and do DFS.
Example

1. Start with an arbitrary vertex and do DFS.
1. Start with an arbitrary vertex and do DFS.
Example

1. Start with an arbitrary vertex and do DFS.
Example

1. Start with an arbitrary vertex and do DFS.
Example

1. Start with an arbitrary vertex and do DFS.

Start:0

Start:1

Start:2

Start:3

Finish:4

1. Start with an arbitrary vertex and do DFS.
Example

1. Start with an arbitrary vertex and do DFS.
Example

1. Start with an arbitrary vertex and do DFS.
1. Start with an arbitrary vertex and do DFS.
Example

1. Start with an arbitrary vertex and do DFS.
Example

1. Start with an arbitrary vertex and do DFS.
1. Start with an arbitrary vertex and do DFS.
1. Start with an arbitrary vertex and do DFS. Repeat until done.
1. Start with an arbitrary vertex and do DFS. Repeat until done.
Example

2. Reverse all the edges.
Example

2. Reverse all the edges.
3. Do DFS again, but this time, start with the vertices with the largest finish time.
Example

3. Do DFS again, but this time, start with the vertices with the largest finish time.

This is one DFS tree in the DFS forest!
3. Do DFS again, but this time, start with the vertices with the largest finish time.
Do DFS again, but this time, start with the vertices with the largest finish time. This is one DFS tree in the DFS forest!

3. Do DFS again, but this time, start with the vertices with the largest finish time.
3. Do DFS again, but this time, start with the vertices with the largest finish time.

Example

This is one DFS tree in the DFS forest!
Example

3. Do DFS again, but this time, start with the vertices with the largest finish time.

Notice that I’m not changing the start and finish times – I’m keeping them from the first run.
3. Do DFS again, but this time, start with the vertices with the largest finish time.

Notice that I’m not changing the start and finish times – I’m keeping them from the first run.
3. Do DFS again, but this time, start with the vertices with the largest finish time.

Notice that I’m not changing the start and finish times – I’m keeping them from the first run.
3. Do DFS again, but this time, start with the vertices with the largest finish time.

Notice that I’m not changing the start and finish times – I’m keeping them from the first run.
3. Do DFS again, but this time, start with the vertices with the largest finish time.
3. Do DFS again, but this time, start with the vertices with the largest finish time.
Example

3. Do DFS again, but this time, start with the vertices with the largest finish time.
3. Do DFS again, but this time, start with the vertices with the largest finish time.
Example

3. Do DFS again, but this time, start with the vertices with the largest finish time.
Example

3. Do DFS again, but this time, start with the vertices with the largest finish time.
3. Do DFS again, but this time, start with the vertices with the largest finish time.
3. Do DFS again, but this time, start with the vertices with the largest finish time.
Shortest path problem

• What is the shortest path between u and v in a weighted graph?
  • the cost of a path is the sum of the weights along that path
  • The shortest path is the one with the minimum cost.

• The distance $d(u,v)$ between two vertices u and v is the cost of the shortest path between u and v.

This path from s to t has cost 25.

This path is shorter, it has cost 5.
Dijkstra
visual intuition
Dijkstra's visual intuition creates a tree!
The highlighted edges along this tree are the fully stretched shortest paths!
Dijkstra Pseudocode

Dijkstra(G,s):

• Set all vertices to **not-sure**
• $d[v] = \infty$ for all $v$ in $V$ (except $s$)
• $d[s] = 0$
• **While** there are **not-sure** nodes:
  • Pick the **not-sure** node $u$ with the smallest estimate $d[u]$.
  • Mark $u$ as **sure**.
  • **For** $v$ in $u$.neighbors:
    • $d[v] \leftarrow \min(d[v], d[u] + \text{edgeWeight}(u,v))$
  • Now $d(s, v) = d[v]$

*Shouldn’t be used on graphs with negative edges!*  
Runtime? Will need to use Fibonacci heaps  
to make operations highlighted operations run faster!
Need to use Fibonacci Heap to implement fastest Dijkstra

<table>
<thead>
<tr>
<th></th>
<th>Sorted Arrays</th>
<th>Linked Lists</th>
<th>Binary Heap</th>
<th>Fibonacci Heap</th>
<th>Red-Black Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search</td>
<td>$O(\log(n))$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(\log(n))$</td>
</tr>
<tr>
<td>Delete</td>
<td>$O(n)$</td>
<td>Search +$O(1)$</td>
<td>Search +$O(\log(n))$</td>
<td>Search +$O(\log(n))$*</td>
<td>$O(\log(n))$</td>
</tr>
<tr>
<td>Insert</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(\log(n))$</td>
<td>$O(1)$</td>
<td>$O(\log(n))$</td>
</tr>
<tr>
<td>Extract-min</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(\log(n))$</td>
<td>$O(\log(n))$*</td>
<td>$O(\log(n))$</td>
</tr>
<tr>
<td>Decrease-key</td>
<td>$O(n)$</td>
<td>Search +$O(1)$</td>
<td>Search +$O(\log(n))$</td>
<td>$O(1)$*</td>
<td>$O(\log(n))$</td>
</tr>
</tbody>
</table>

* - amortized time
Dijkstra with Fibonacci Heap

- \( T(\text{extractMin}) = O(\log(n)) \)  \((\text{amortized time}^*)\)
- \( T(\text{decreaseKey}) = O(1) \)  \((\text{amortized time}^*)\)
- See CS166 for more! (or CLRS)

Running time of Dijkstra

\[
= O(n \cdot T(\text{extractMin}) + m \cdot T(\text{decreaseKey})) \\
= O(n \log(n) + m)
\]

*This means that any sequence of \(d\) extractMin calls takes time at most \(O(d\log(n))\). But a few of the \(d\) may take longer than \(O(\log(n))\) and some may take less time.
What about a Dynamic Programming approach to shortest path problems?
Assumption: no negative cycles

• Negative weights are possible (some algorithms won’t work in that case!)
  • Example: negative costs because I pick up some passengers

• But if we have negative cycles…

• Shortest paths aren’t defined if there are negative cycles!
Shortest path using DP

• Step 1:

**Optimal substructure**: shortest path using \( \leq i \) edges

• Step 2:
Suppose we already know \( d^i(s,u) \) for fixed \( s \) and all \( u \)

**Recursive formulation**: 
\[
d^{i+1}(s,v) = \min_{u} \{d^i(s,u) + w(u,v)\}
\]
Step 3: write the algorithm

Bellman-Ford(G,s):

- \( d^{(0)}(v) = \infty \) for all \( v \) in \( V \)  
  // initialize: \( d^{(i)}(v) \) is distance from \( s \) to \( v \) with \( \leq i \) edges
- \( d^{(0)}(s) = 0 \)

- For \( i=0,\ldots,n-2 \):
  - \( d^{(i+1)}(v) = d^{(i)}(v) \) for all \( v \) in \( V \)  
    // baseline distance: \( v \) doesn’t need \((i+1)^{th}\) edge
  - For \( v \) in \( V \):
    - For \( u \) in \( v \).neighbors:
      - \( d^{(i+1)}(v) \leftarrow \min(d^{(i+1)}(v), d^{(i)}(u) + \text{edgeWeight}(u,v)) \)
        // found a better path through \( u \)

- Return \( d^{(n-1)} \)
Bellman-Ford take-aways

• Running time is $O(mn)$
  • For each of $n$ rounds, update $m$ edges.

• For $i=0,\ldots,n-1$:
  • For $u$ in $V$:
    • For $v$ in $u$.neighbors:

• Works fine with negative edges.
• Does not work with negative cycles.
  • But it can detect negative cycles!
Note on implementation

• Don’t actually keep all n arrays around.
• Just keep two at a time: “last round” and “this round”
Thank you!