Heaps

- What’s an abstract data type?
- How can we implement a heap?
  - Build-Heap
  - Extract-Max
  - Insert

Proofs on trees
Hotels!

Topological sort
A linear ordering of all the vertices such that for all edges \((u,v)\)
- \(u\) appears before \(v\) in the ordering

An ordering of the nodes that "obeys" the dependencies, i.e.,
an activity can't happen until its dependent activities have
happened

DAGs
Can represent dependency graphs

Topological sort
- Topological-Sort1\((G)\)
  1. Find a node \(v\) with no incoming edges
  2. Delete \(v\) from \(G\)
  3. Add \(v\) to linked list
  4. Topological-Sort1\((G)\)
Topological sort

1. Find a node $v$ with no incoming edges
2. Delete $v$ from $G$
3. Add $v$ to linked list
4. Topological-Sort1($G$)

Topological-Sort1($G$)

1. Find a node $v$ with no incoming edges
2. Delete $v$ from $G$
3. Add $v$ to linked list
4. Topological-Sort1($G$)
Topological sort

1. Find a node \( v \) with no incoming edges
2. Delete \( v \) from \( G \)
3. Add \( v \) to linked list
4. \textsc{Topological-Sort}(G)

Topological sort

1. Find a node \( v \) with no incoming edges
2. Delete \( v \) from \( G \)
3. Add \( v \) to linked list
4. \textsc{Topological-Sort}(G)
Running time?

**Topological-Sort1(G)**
1. Find a node \( v \) with no incoming edges
2. Delete \( v \) from \( G \)
3. Add \( v \) to linked list
4. **Topological-Sort1(G)**

O(|V|+|E|)

Running time?

**Topological-Sort1(G)**
1. Find a node \( v \) with no incoming edges
2. Delete \( v \) from \( G \)
3. Add \( v \) to linked list
4. **Topological-Sort1(G)**

O(|V|+|E|)

Running time?

**Topological-Sort1(G)**
1. Find a node \( v \) with no incoming edges
2. Delete \( v \) from \( G \)
3. Add \( v \) to linked list
4. **Topological-Sort1(G)**

O(E) overall

Running time?

**Topological-Sort1(G)**
1. Find a node \( v \) with no incoming edges
2. Delete \( v \) from \( G \)
3. Add \( v \) to linked list
4. **Topological-Sort1(G)**

How many calls? \(|V|\)
Running time?

**Topological-Sort1(G)**
1. Find a node \(v\) with no incoming edges
2. Delete \(v\) from \(G\)
3. Add \(v\) to linked list
4. **Topological-Sort1(G)**

Overall running time?

\[O(|V|^2 + |V| |E|)\]

Can we do better?

**Topological-Sort1(G)**
1. Find a node \(v\) with no incoming edges
2. Delete \(v\) from \(G\)
3. Add \(v\) to linked list
4. **Topological-Sort1(G)**

Topological sort 2

**Topological-Sort2(G)**
1. for all edges \((u, v) \in E\)
2. \(active[u] \leftarrow active[u] + 1\)
3. for all \(v \in V\)
4. if \(active[v] = 0\)
5. \(ENQUEUE(S, v)\)
6. while \(!EMPTY(S)\)
7. \(u \leftarrow DEQUEUE(S)\)
8. add \(u\) to linked list
9. for each edge \((u, v) \in E\)
10. \(active[v] \leftarrow active[v] - 1\)
11. if \(active[v] = 0\)
12. \(ENQUEUE(S, v)\)

Topological sort 2

**Topological-Sort2(G)**
1. for all edges \((u, v) \in E\)
2. \(active[u] \leftarrow active[u] + 1\)
3. for all \(v \in V\)
4. if \(active[v] = 0\)
5. \(ENQUEUE(S, v)\)
6. while \(!EMPTY(S)\)
7. \(u \leftarrow DEQUEUE(S)\)
8. add \(u\) to linked list
9. for each edge \((u, v) \in E\)
10. \(active[v] \leftarrow active[v] - 1\)
11. if \(active[v] = 0\)
12. \(ENQUEUE(S, v)\)
### Topological sort 2

**Algorithm TopologicalSort2(G)**

1. for all edges \( (u, v) \in E \)
2. \( \text{active}[v] \leftarrow \text{active}[v] + 1 \)
3. for all \( v \in V \)
4. \[\text{if active}[v] = 0 \]
5. \[\text{ENQUEUE}(S, v)\]
6. while !\text{EMPTY}(S)
7. \[u \leftarrow \text{DEQUEUE}(S)\]
8. add \( u \) to linked list
9. for each edge \( (u, v) \in E \)
10. \[\text{active}[v] \leftarrow \text{active}[v] - 1\]
11. \[\text{if active}[v] = 0\]
12. \[\text{ENQUEUE}(S, v)\]

### Running time?

<table>
<thead>
<tr>
<th>How many times do we process each node?</th>
</tr>
</thead>
<tbody>
<tr>
<td>How many times do we process each edge?</td>
</tr>
</tbody>
</table>

**Time Complexity:** \(O(|V| + |E|)\)

### Detecting cycles

#### Undirected graph
- **BFS or DFS:** If we reach a node we've seen already, then we've found a cycle.

#### Directed graph
- **Call TopologicalSort**
- **If the length of the list returned \( \neq |V| \) then a cycle exists**
**Connectedness**

Given an undirected graph, for every node $u \in V$, can we reach all other nodes in the graph?

**Algorithm + running time**

- Run BFS or DFS-Visit (one pass) and mark nodes as we visit them. If we visit all nodes, return true, otherwise false.

- Running time: $O(|V| + |E|)$


**Strongly connected**

Given a directed graph, can we reach any node $v$ from any other node $u$?

Can we do the same thing?


**Transpose of a graph**

Given a graph $G$, we can calculate the transpose of a graph $G^R$ by reversing the direction of all the edges.

- Running time to calculate $G^R$: $O(|V| + |E|)$


**Strongly connected**

Strongly-Connected($G$)

- Run DFS-Visit or BFS from some node $u$
- If not all nodes are visited: return false
- Create graph $G^t$
- Run DFS-Visit or BFS on $G^t$ from node $u$
- If not all nodes are visited: return false
- return true
Is it correct?

What do we know after the first pass?
- Starting at u, we can reach every node.

What do we know after the second pass?
- All nodes can reach u. Why?
  - We can get from u to every node in G, therefore, if we reverse the edges (i.e. G), then we have a path from every node to u.

Which means that any node can reach any other node. Given any two nodes s and t we can create a path through u.

Runtime?

Strongly-Connected(G)
- Run DFS-Visit or BFS from some node u
- If not all nodes are visited: return false
- Create graph G^R
- Run DFS-Visit or BFS on G^R from node u
- If not all nodes are visited: return false
- return true

O(|V| + |E|)

Shortest paths

Dijkstra’s

Bellman-Ford

Floyd-Warshall

Johnson’s