CS 61B Spring 2026
Heaps, Tree Traversals, and Graphs
Exam-Level Discussion 09: March 16, 2025

1 Heaps and Graphs Summary

Heap. A min-heap is a complete binary tree where every node is smaller than its children. When used to implement a Priority Queue, then operations are as follows:

- **Add:** Add to the end of the array, then *swim up* (repeatedly swap with parent while smaller). Runtime: $O(\log N)$.
- **Remove Smallest:** Swap root with last element, remove last, then *sink down* (repeatedly swap with the smaller child while larger). Runtime: $O(\log N)$.

Tree Traversals. Suppose we take some action on every node of a tree, e.g. printing. Suppose current node is v, left subtree L, and right subtree R:

Pre-order:	action(v), L, R
In-order:	L, action(v), R
Post-order:	L, R, action(v)
Level-order:	Top to bottom, left to right

Graphs. A graph $G = (V, E)$ consists of vertices V and edges E. Representations:
- **Adjacency list:** For each vertex, store a list of its neighbors. Space: $\Theta(V + E)$.
- **Adjacency matrix:** $V \times V$ grid; entry $(i, j) = 1$ if edge exists. Space: $\Theta(V^2)$.

Graph Traversals.
- **DFS (Depth-First Search):** Recursively explore vertices by going as deep as possible before backtracking. Use a `marked` array to avoid infinite recursion. Runtime: $\Theta(V + E)$.

```
dfs(Graph G, int v) {
  marked[v] = true;                   // LINE 1
  for (int w : G.adj(v)) {
      if (!marked[w]) {               // LINE 2
          edgeTo[w] = v;
          dfs(G, w);
      }
  }
```

  - DFS Pre-order: The order in which DFS calls are made.
  - DFS Post-order: The order in which DFS calls complete.
- **BFS (Breadth-First Search):** Visit vertices in order of number of edges from source. Uses a `Queue`. Runtime: $\Theta(V + E)$. Covered on next worksheet.
- **Dijkstra's (Best-First Search):** Visit vertices in order of total distance from source (based on edge weights). Uses a `Priority Queue`. Runtime: $\Theta((V + E) \log V)$. Covered on next worksheet.

2 Heap Mystery

We are given the following array representing a min-heap where each letter represents a **unique** number. Assume the root of the min-heap is at index one, i.e. **A** is the root. Our task is to figure out the numeric ordering of the letters. Therefore, there is **no** significance of the alphabetical ordering. i.e. just because B precedes C in the alphabet, we do not know if B is less than or greater than C.

Array: [-, A, B, C, D, E, F, G]

Four unknown operations are then executed on the min-heap. An operation is either a **removeMin** or an **insert**. The resulting state of the min-heap is shown below.

Array: [-, A, E, B, D, X, F, G]

(a) Determine the operations executed and their appropriate order. The first operation has already been filled in for you!

Hint: Which elements are gone? Which elements are newly added? Which elements are removed and then added back?

1. `removeMin()`
2. `insert(X)`
3. `removeMin()`
4. `insert(A)`

Explanation: We know immediately that A was removed. Then, after looking at the final state of the min-heap, we see that C was removed. Then, for A to remain in the min-heap, we see that A must have been inserted afterwards. And, after seeing a new value X in the min-heap, we see that X must have been inserted as well. We just need to determine the relative ordering of the **insert(X)** in between the operations **removeMin()** and **insert(A)**, and we see that the **insert(X)** must go before both.

(b) Fill in the following comparisons with either >, <, or ? if unknown. We recommend considering which elements were compared to reach the final array.

1. X ? D
2. X > C
3. B > C
4. G < X

Reasoning:
1. X is never compared to D
2. X must be greater than C since C is removed after X's insertion.
3. B must also be greater than C otherwise the second call to **removeMin** would have removed B
4. X must be greater than G so that it can be "promoted" to the top after the removal of C. It needs to be promoted to the top to land in its new position.

3 Graph Conceptuals

(a) Answer the following questions as either **True** or **False** and provide a brief explanation:

1. If a graph with n vertices has $n-1$ edges, it **must** be a tree.

   Solution: False. The graph **must** be connected.

2. Every edge is looked at exactly twice in each full run of DFS on a connected, undirected graph.

   Solution: True. Say an edge connects u and v. Both u and v will look at the other one through this edge when it's their turn.

3. In BFS, let $d(v)$ be the minimum number of edges between a vertex v and the start vertex. For any two vertices u, v in the fringe (recall that the fringe in BFS is a queue), $|d(u) - d(v)|$ is **always less than** 2.

   Solution: True. Suppose this was not the case. Then, we could have a vertex 2 edges away and a vertex 4 edges away in the fringe at the same time. But, the only way to have a vertex 4 edges away is if a vertex 3 edges away was removed from the fringe. We see this could never occur because the vertex 2 edges away would be removed before the vertex 3 edges away!

(b) Given an undirected graph, provide an algorithm that returns true if a cycle exists in the graph, and false otherwise. Also, provide a Θ bound for the worst case runtime of your algorithm.

Solution: We do a depth first search traversal through the graph. While we recurse, if we visit a node that we visited already, then we've found a cycle. Assuming integer labels, we can use something like a `visited` boolean array to keep track of the elements that we've seen, and while looking through a node's neighbors, if `visited` gives true, then that indicates a cycle.

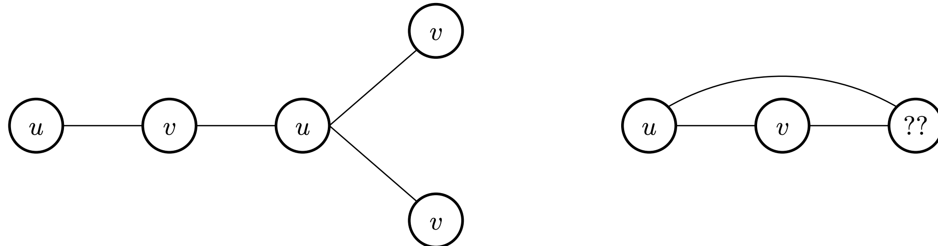
However, since the graph is undirected, if an edge connects vertices `u` and `v`, then `u` is a neighbor of `v`, and `v` is a neighbor of `u`. As such, if we visit `v` after `u`, our algorithm will claim that there is a cycle since `u` is a visited neighbor of `v`. To address this case, when we visit the neighbors of `v`, we should ignore `u`. To implement this in code, we could add the parent as another parameter in the method call. In the worst case, we have to explore at most V edges before finding a cycle (number of edges doesn't matter). So, this runs in $\Theta(V)$.

Pseudocode is provided below (for a disconnected graph, we should call `find_cycle` on each component).

```
find_cycle(v, parent=-1):
    visited[v] = true
    for (v, w) in G:
        if !visited[w]:
            if find_cycle(w, v):
                return True
        else if w != parent:
            return True
    return False
```

4 Graph Algorithm Design

(a) An undirected graph is said to be bipartite if all of its vertices can be divided into two disjoint sets U and V such that every edge connects an item in U to an item in V. For example below, the graph on the left is bipartite, whereas on the graph on the right is not. Provide an algorithm which determines whether or not a graph is bipartite. What is the runtime of your algorithm?
Hint: Can you modify an algorithm we already know (ie. graph traversal)?



Solution:

To solve this problem, we run a special version of a traversal from any vertex. This can be implemented using either DFS and BFS as the underlying traversal that we will modify. Our special version marks the start vertex with a u, then each of its neighbors with a v, and each of their neighbors with a u, and so forth. If at any point in the traversal we want to mark a node with u but it is already marked with a v (or vice versa), then the graph is not bipartite.

If the graph is not connected, we repeat this process for each connected component.

If the algorithm completes, successfully marking every vertex in the graph, then it is bipartite.

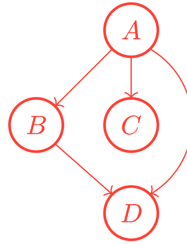
The runtime of the algorithm is the same whether you use BFS or DFS: $\Theta(E + V)$.

(b) Consider the following implementation of DFS, which contains a crucial error:

```
create the fringe, which is an empty Stack
push the start vertex onto the fringe and mark it
while the fringe is not empty:
    pop a vertex off the fringe and visit it
    for each neighbor of the vertex:
        if neighbor not marked:
            push neighbor onto the fringe
            mark neighbor
```

First, identify the bug in this implementation. Then, give an example of a graph where this algorithm may not traverse in DFS order.

Solution:



For the graph above, it's possible to visit in the order $A-B-C-D$ (which is not depth-first) because D won't be put into the fringe after visiting B, since it's already been marked after visiting A. One should only mark nodes when they have actually been visited, but in this buggy implementation, we mistakenly mark them before we visit them, as we're putting them into the fringe.

(c) *Extra:* Provide an algorithm that finds the shortest cycle (in terms of the number of edges used) in a directed graph in $O(EV)$ time and $O(E)$ space, assuming $E > V$.

Solution:

The key realization here is that the shortest directed cycle involving a particular source vertex s is just the shortest path to a vertex v that has an edge to s, along with that edge. Using this knowledge, we create a **`shortestCycleFromSource(s)`** subroutine. This subroutine runs BFS on s to find the shortest path to every vertex in the graph. Afterwards, it iterates through all the vertices to find the shortest cycle involving s: if a vertex v has an edge back to s, the length of the cycle involving s and v is one plus **`distTo`**(v) (which was computed by BFS).

An alternative approach to the above subroutine (that is slightly more optimized) actually modifies BFS to short circuit if it's visiting a node v and sees it has an edge v -> s. Because BFS visits in order of distance from s, we can know that the first vertex v we see with an edge back to s will be the shortest cycle.

Regardless of which approach you take, asymptotically our subroutine takes $O(E+V)$ time because it uses BFS and a linear pass through the vertices. To find the shortest cycle in an entire graph, we simply call the subroutine on each vertex, resulting in an $V \cdot O(E+V) = O(EV + V^2)$ runtime. Since $E > V$, this is still $O(EV)$, since $O(EV + V^2) \in O(EV + EV) \in O(EV)$.