CS 61B
Spring 2025

Graphs II, Tries
Discussion 10: April 07, 2025

1 Longest Prefix

Fill in the `longestPrefixOf(String word)` method below such that it returns the longest prefix of `word` that is also a prefix of a key in the trie.

For example, if a `TrieSet t` contains keys `{"cryst", "tries", "cr"}`, then `t.longestPrefixOf("crystal")` returns `"cryst"` and `t.longestPrefixOf("crys")` returns `"crys"`.

The code uses the `StringBuilder` class to build strings character-by-character. To add a character to the end of the `StringBuilder`, use the `append(char c)` method. Once all characters have been appended, the resulting String is returned by the `toString()` method.

```
  StringBuilder sb = new StringBuilder();
  sb.append('a');
  sb.append('b');
  System.out.println(sb.toString()); // "ab"
```

```
public class TrieSet {
    private Node root;
    private class Node {
        boolean isKey;
        Map<Character, Node> map;
        private Node() {
            isKey = false;
            map = new HashMap<>();
        }
    }

    public String longestPrefixOf(String word) {
        int n = word.length();
        StringBuilder prefix = new StringBuilder();
        Node curr = _________________________;
        for (_____________________________________________) {
            _____________________________________________________
            _____________________________________________________
            _____________________________________________________
            _____________________________________________________
            _____________________________________________________
            _____________________________________________________
        }
        return _____________________________________________
    }
}
```

Solution:

```
public String longestPrefixOf(String word) {
    int n = word.length();
    StringBuilder prefix = new StringBuilder();
    Node curr = root;
    for (int i = 0; i < n; i++) {
        char c = word.charAt(i);
        if (!curr.map.containsKey(c)) {
            break;
        }
        curr = curr.map.get(c);
        prefix.append(c);
    }
    return prefix.toString();
}
```

2 A Tree Takes On Graphs

Your friend at Stanford has come to you for help on their homework! For each of the following statements, determine whether they are true or false; if false, provide counterexamples.

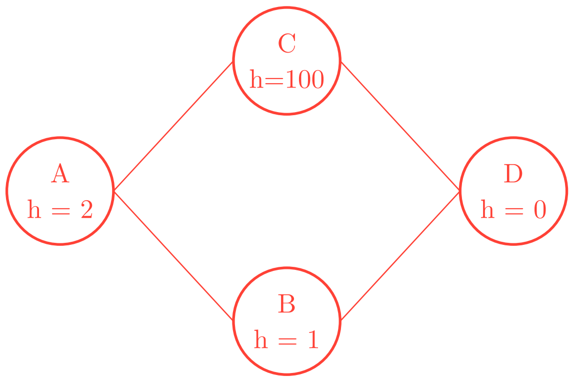
(a) "A graph with edges that all have the same weight will always have multiple MSTs."

Solution: False: Consider a tree (N nodes and N - 1 edges, no cycles, connected) - there is only one way to connect a tree, so it will be its own MST (so there is only one MST for a tree).

(b) "No matter what heuristic you use, A* search will always find the correct shortest path."

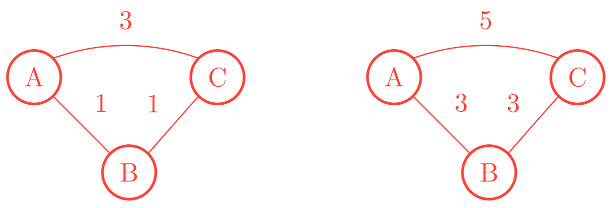
Solution: False: Here, A* would incorrectly return A - B - D as the shortest path from A to D. Starting at A, we would add B to the queue with priority 1+1 (the known distance to B, as well as our estimated distance from B to the goal), and we would add C to the queue with priority 1+100 (the known distance to C, as well as our estimated distance from C to the goal). We then pop B off the queue, and add D to the queue with priority 11+0 (the known distance to D, as well as the estimated distance from D to the goal). Our queue now contains C with priority 101, and D with priority 11, so we pop D off the queue and complete our search, returning A - B - D as the shortest path instead of the correct answer: A - C - D.

In general, A* is only guaranteed to be correct if the heuristic is good-specifically, it should be both admissible and consistent (note that applying admissibility and consistency are out of scope for this class, you only need to know the definition). In the example given, our heuristic is neither admissible nor consistent.



(c) "If you add a constant factor to each edge in a graph, Dijkstra's algorithm will return the same shortest paths tree."

False: this can be disproved with the example below, where we add a constant $c = 2$ to every edge. Adding a constant factor per edge will disadvantage paths with more edges. In our example, though A - B - C had more edges, in our original graph it still has shorter total path cost, at $1 + 1 = 2$. On the other hand, A - C had fewer edges but larger total path cost, 3. After adding a constant factor, however, the A - B - C path cost was $(1 + 2) + (1 + 2) = 6$ and the A - C had a path cost of $(2 + 3) = 5$. So before the addition, Dijkstra's shortest path tree would have said the shortest path from A to C was A - B - C, but afterwards it would say A - C.



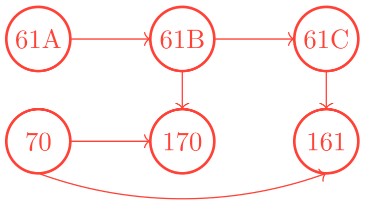
3 Class Enrollment

You're planning your CS classes for the upcoming semesters, but it's hard to keep track of all the prerequisites! Let's figure out a valid ordering of the classes you're interested in. A valid ordering is an ordering of classes such that every prerequisite of a class is taken before the class itself. Assume we're taking one CS class per semester.

(a) The list of prerequisites for each course is given below (not necessarily accurate to actual courses!). Draw a graph to represent our scenario.

- CS 61A: None
- CS 61B: CS 61A
- CS 61C: CS 61B
- CS 70: None
- CS 170: CS 61B, CS 70
- CS 161: CS 61C, CS 70

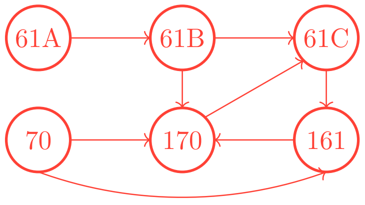
Solution:



(b) Suppose we added a new prerequisite where the student must take CS 161 before CS 170 and CS 170 before CS 61C. Is there still a valid ordering of classes such that no prerequisites are broken? If no, explain.

Solution:

The new graph looks like this:



There exists a cycle between 161 → 170 → 61C → 161, so a valid ordering does not exist. Our graph must be directed and acyclic for a topological sort to work.

(c) With the original graph, perform a topological sort to find a valid ordering of the 6 classes. Break ties by going to the lower course number first.

Solution:

With topological sorting, if an edge from vertex u to vertex v exists in the graph, then u must come before v in the sorted order. Every edge in our graph represents a prerequisite where class u must be taken before v. So, our topologically sorted order will ensure that we meet all prerequisites!

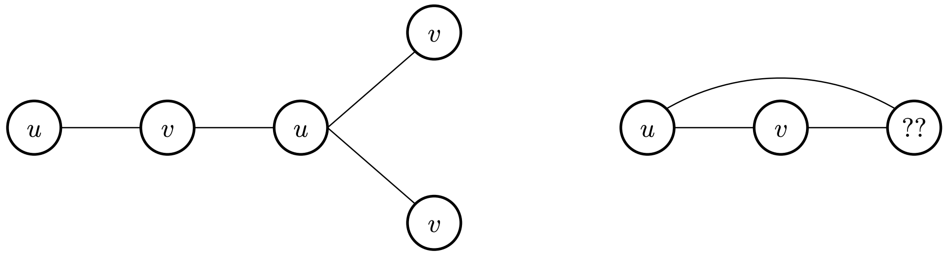
To topological sort on a graph, perform DFS from each vertex with indegree 0 (no incoming edges), but don't clear the node's we've marked between each new DFS traversal. Afterwards, we reverse the postorder to get our topologically sorted order.

In our graph, there are two vertices with indegree 0: CS 61A and CS 70. If we DFS from CS 61A then CS 70, we get a postorder of: [CS 161, CS 61C, CS 170, CS 61B, CS 61A, CS 70]. Reversing this order gives a valid ordering of: [**CS 70, CS 61A, CS 61B, CS 170, CS 61C, CS 161**].

If we DFS from CS 70 then CS 61A, we get a postorder of: [CS 161, CS 170, CS 70, CS 61C, CS 61B, CS 61A]. Reversing this order gives a different but still valid ordering of: [**CS 61A, CS 61B, CS 61C, CS 70, CS 170, CS 161**].

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.

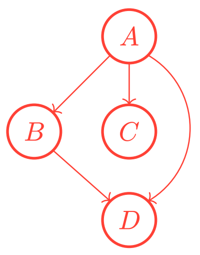
The runtime of the algorithm is the same whether you use BFS or DFS: $\Theta(E + V)$.

(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 $O(EV+V^2) \in O(EV+EV) \in O(EV)$.