CS 61B
Fall 2025

MSTs, DAGs, and Hashing

Exam-Level Discussion 09: October 27, 2025

1 Multiple MSTs

Recall a graph can have multiple MSTs if there are multiple spanning trees of minimum weight.

(a) For each subpart below, select the correct option and justify your answer. If you select "never" or "always," provide a short explanation. If you select "sometimes," provide two graphs that fulfill the given properties.

1. If **some** of the edge weights are **identical**, there will

   ○ never be multiple MSTs in G.

   ○ sometimes be multiple MSTs in G.

   ○ always be multiple MSTs in G.

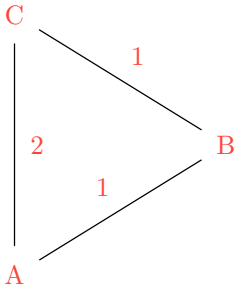
   Justification:

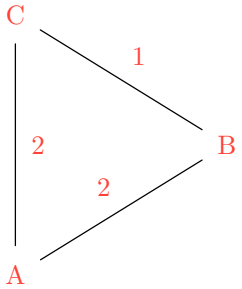
   Solution: If **some** of the edge weights are **identical**, there will

   ○ never be multiple MSTs in G.

   ● sometimes be multiple MSTs in G.

   ○ always be multiple MSTs in G.





   Justification:

   In the graph on the left, the only MST is [AB, BC]. In the graph on the right, two MSTs exist — [AB, BC] and [AC, BC].

2. If **all** of the edge weights are **identical**, there will

   ○ never be multiple MSTs in G.

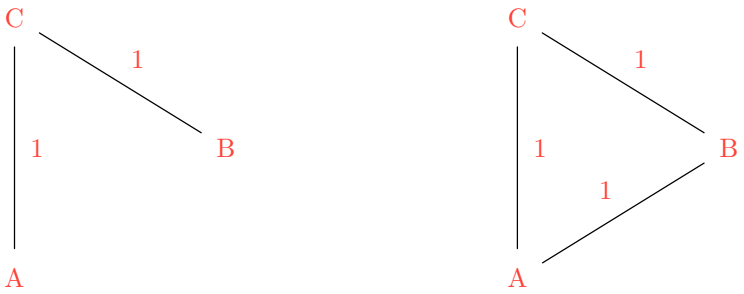
   ○ sometimes be multiple MSTs in G.

   ○ always be multiple MSTs in G.

   Justification:

Solution: If **some** of the edge weights are **identical**, there will

- ○ never be multiple MSTs in G.
- ● sometimes be multiple MSTs in G.
- ○ always be multiple MSTs in G.



Justification:

In the graph on the left, the only MST is [AB, AC]. Note that for any tree, we only have one MST, since the tree itself is the MST! In the graph on the right, three MSTs exist — [AB, BC], [AC, BC], and [AB, AC].

(b) Suppose we have a connected, undirected graph (G) with (N) vertices and (N) edges, where all the **edge weights are identical**. Find the maximum and minimum number of MSTs in (G) and explain your reasoning.

Minimum:

Maximum:

Justification:

Solution:

Minimum: 3

Maximum: *N*

Justification: Notice that if all the edge weights are the same, an MST is just a spanning tree. Let's begin by creating a tree, i.e. a connected graph with *N−1* edges. Now, notice that there is only one spanning tree, since the graph is itself a tree.

As such, the problem reduces to: how many spanning trees can the insertion of one edge create? If we add an edge to a tree, it will create a cycle that can be of length at minimum 3 and at maximum *N*. Then, notice that we can only remove **any** edge from a cycle to create a spanning tree, so we have at minimum 3 and at maximum *N* possible MSTs in G.

(c) It is possible that Prim's and Kruskal's find different MSTs on the same graph G (as an added exercise, construct a graph where this is the case!).

Given any graph G with integer edge weights, modify the edge weights of G to *ensure* that

(1) Prim's and Kruskal's will output the same results, and

(2) the output edges still form a MST correctly in the original graph.

You may not modify Prim's or Kruskal's, and you may not add or remove any nodes/edges.

Hint: Look at subpart 1 of part (a).

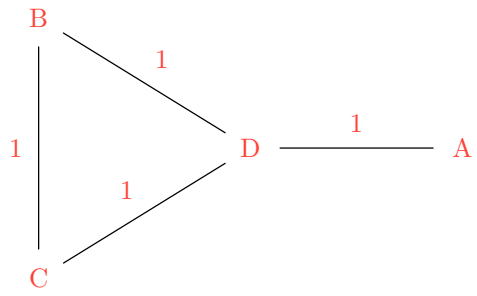
Solution:

To ensure that Prim's and Kruskal's will always produce the same MST, notice that if G has unique edges, only one MST can exist, and Prim's and Kruskal's will always find that MST! So, what if we modify G to ensure that all the edge weights are unique?

To achieve this, let's strategically add a small, unique offset between 0 and 1, exclusive, to each edge. It is important that we choose an offset between 0 and 1. This is to ensure that the edges picked in the modified graph is still a correct MST in the original graph, since all the edge weights are integers. It is also important that the offset is unique for each edge, because then we ensure each weight is distinct. Pseudocode for such a change is shown below:

```
E = number of edges in the graph
    offset = 0
    for edge in graph:
        edge.weight += offset
        offset += 1 / E
```

In regard to the added exercise, here is a simple graph G where Prim's and Kruskal's produce different MSTs. Prim's starting from A will select AD, BD, and CD, whereas Kruskals will select AD, BC, and BD.



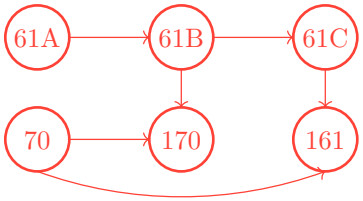
2 Class Enrollment

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

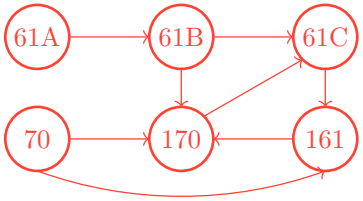
Solution:



(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**].

3 Hashing Gone Crazy

For this question, use the following `TA` class for reference.

```
public class TA {
    int semester;
    String name;
    TA(String name, int semester) {
        this.name = name;
        this.semester = semester;
    }
    @Override
    public boolean equals(Object o) {
        TA other = (TA) o;
        return other.name.charAt(0) == this.name.charAt(0);
    }
    @Override
    public int hashCode() { return semester; }
}
```

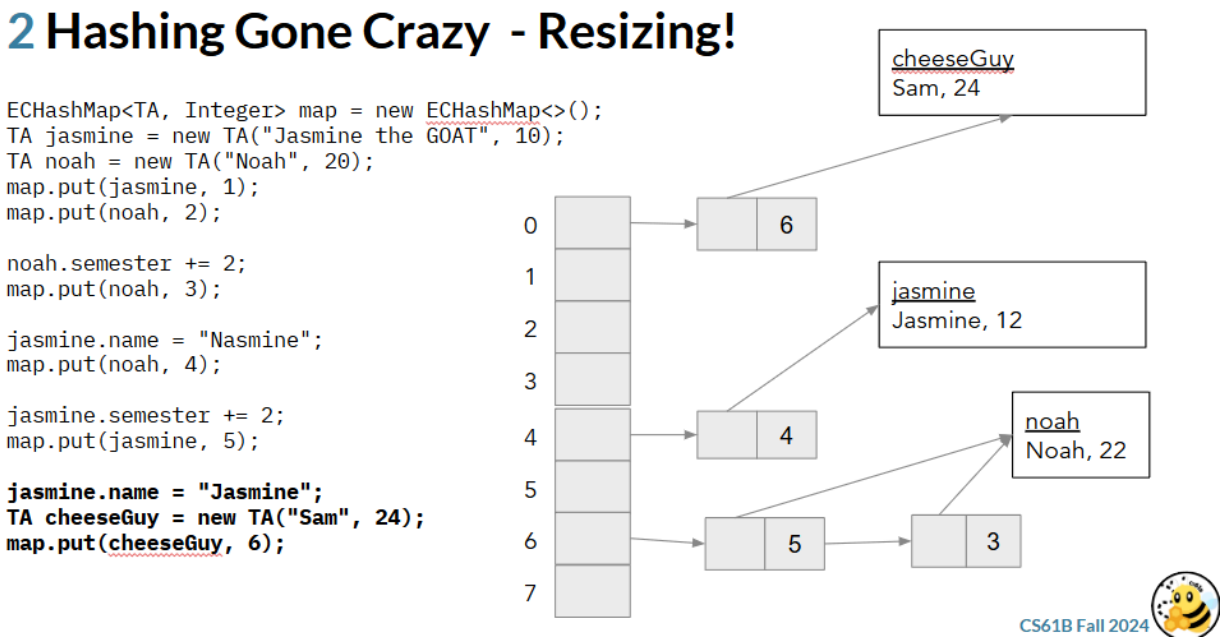
Assume that the `ECHashMap` is a `HashMap` implemented with **external chaining** as depicted in lecture. The `ECHashMap` instance begins with 4 buckets.

Resizing Behavior If an insertion causes the load factor to reach or exceed 1, we resize by doubling the number of buckets. During resizing, we traverse the linked list that correspond to bucket 0 to rehash items one by one, and then traverse bucket 1, bucket 2, and so on. Duplicates are **not** checked when rehashing into new buckets.

Draw the contents of map after the executing the insertions below:

```
ECHashMap<TA, Integer> map = new ECHashMap<>();
TA jasmine = new TA("Jasmine the GOAT", 10);
TA noah = new TA("Noah", 20);
map.put(jasmine, 1);
map.put(noah, 2);

noah.semester += 2;
map.put(noah, 3);

jasmine.name = "Nasmine";
map.put(noah, 4);

jasmine.semester += 2;
map.put(jasmine, 5);

jasmine.name = "Jasmine";
TA cheeseguy = new TA("Sam", 24);
map.put(cheeseguy, 6);
```

Solution:



Explanation:

Line 4: `jasmine` has `semester` value 10. `10 % 4 = 2`, so `jasmine` is placed in bucket 2 with value 1.
`0: [], 1: [], 2: [(jasmine, 1)], 3: []`

Line 5: `noah` is placed in bucket 0 with value 2.
`0: [(noah, 2)], 1: [], 2: [(jasmine, 1)], 3: []`

Line 7: Increasing the `semester` value of `noah` does *not* cause it to be rehashed! (This is why modifying objects in a Hashmap is dangerous - it can change the hashcode of your object and make it impossible to find which bucket it belongs to).

Line 8: `noah` now has `semester` 4, so bucket 2 also has a node pointing to `noah`, with value 3. (Note that the two `noahs` refer to the same object).
`0: [(noah, 2)], 1: [], 2: [(jasmine, 1), (noah, 3)], 3: []`

Line 11, 12: `noah` with `semester` 22 hashes to bucket 2. However, since we have changed `jasmine`'s name to be "`Nasmine`", `noah.equals(jasmine)` returns `true`. Since we are hashing a key that is already present in the dictionary according to `.equals`, we replace `jasmine`'s old value with the new value, 4.
`0: [(noah, 2)], 1: [], 2: [(jasmine, 4), (noah, 3)], 3: []`

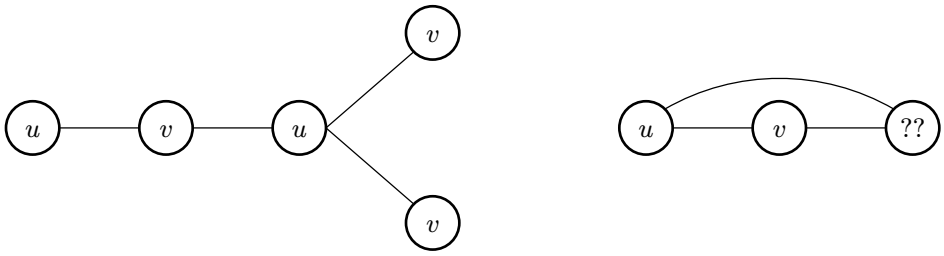
Line 13, 14: `jasmine` with `semester` 12 hashes to bucket 0. However, since we have changed `jasmine`'s name to be "Nasmine", `jasmine.equals(noah)` returns `true`. Since we are hashing a key that is already present in the dictionary according to `.equals`, we replace `noah`'s old value with the new value, 5.
`0: [(noah, 5)], 1: [], 2: [(jasmine, 4), (noah, 3)], 3: []`

Line 16, 17, 18: `cheeseGuy` hashes to bucket 0. `cheeseGuy.equals(noah)` returns `false`, so we add a new node after `noah` with value 6.
`0: [(noah, 5), (cheeseGuy, 6)], 1: [], 2: [(jasmine, 4), (noah, 3)], 3: []`

Resizing: The load factor reaches 1, and we resize to 8 buckets. We rehash the elements in the order they were inserted. Notice that duplicates are not checked when rehashing into new buckets, therefore, the `noah` object is inserted twice.

4 Extra: 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.

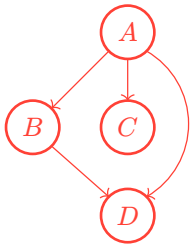
The runtime of the algorithm is the same whether you use BFS or DFS: $\Theta(E + V)$.

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

5 Extra: Buggy Hash

The following classes may contain a bug in one of its methods. Identify those errors and briefly explain why they are incorrect and in which situations would the bug cause problems.

(a) The `Timezone` class below:

```
class Timezone {
    String timeZone; // "PST", "EST" etc.
    boolean dayLight;
    String location;
    ...
    public int currentTime() {
        // return the current time in that time zone
    }

    public int hashCode() {
        return currentTime();
    }

    public boolean equals(Object o) {
        Timezone tz = (Timezone) o;
        return tz.timeZone.equals(timeZone);
    }
}
```

Solution: Although equal objects will have the same hashcode, but the problem here is that `hashCode()` is not deterministic. This may result in weird behaviors (e.g. the element getting lost) when we try to put or access elements.

(b) The `Course` class below:

```
class Course {
    int courseCode;
    int yearOffered;
    String[] staff;
    ...
    public int hashCode() {
        return yearOffered + courseCode;
    }

    public boolean equals(Object o) {
        Course c = (Course) o;
        return c.courseCode == courseCode;
    }
}
```

Solution: The problem with this `hashCode()` is that not all equal objects have the same hashcode. This may produce unexpected behavior, e.g. multiple "equal" objects may be exist in different buckets in the HashMap, the `containsKey` operation may return false, etc. One key thing to remember is that when we override the `equals()` method, we have to also override the `hashCode()` method to ensure equal objects have the same hashCode.