Problem Set 4

All parts are due April 7 at 11:59PM. Please download the .zip archive for this problem set. Remember, your goal is to communicate. Full credit will be given only to a correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem.

Part A

Problem 4-1. [20 points] **Hashing with Deletion**

In this problem, we consider hash tables that support Insert, Delete, and Search operations. We consider both tables that use chaining and tables that use open addressing. In all the cases we consider, the data structure will contain no more than one entry for any particular key k. The three operations are specified as follows.

- $Insert(k)$ adds an entry for key k to the data structure if k does not already appear in the structure, and returns a pointer to it. If k is already there, the operation returns "already there".
- $Delete(k)$ removes the entry for k from the data structure if k appears in the structure. If not, the operation returns "not there".
- $Search(k)$ returns a pointer to the entry for key k if such an entry appears in the data structure. If not, it returns "not there".

In the following problem parts, you may assume that the load factor of the table is small, so you need not worry about handling table overflow.

(a) [5 points] Consider hashing with chaining, based on a hash function $h(k)$. Describe (in words and pseudocode) an algorithm to implement the hash table. That is, describe the data structure, its initialization, and the algorithms for $Insert$, $Delete$, and $Search$.

Solution: We assume for convenience in the pseudocode that the chains are doubly-linked lists with "next" and "prev" fields. (Maintaining both pointers does not affect the asymptotic complexity.) We assume the last node in the linked list contains the value NIL in its "next" field.

Everywhere in this code, the & operator returns the pointer associated with a particular value.

$Insert(k)$: From location $h(k)$, traverse the associated linked list looking for k until you find it or reach the end. If you find it, return "already there". Otherwise, add a k entry at the end and return a pointer to the entry.

INSERT-WITH-CHAINING$(table, k)$

```
1  linkedlist_node = table[h(k)].head
2  while linkedlist_node.next ≠ NIL
3          if linkedlist_node.key = k
4                  return "already there"
5          linkedlist_node = linkedlist_node.next
6  if linkedlist_node.key = k
7          return "already there"
8  linkedlist_node.next = (k, linkedlist_node, NIL)
9  return &linkedlist_node.next
```

$Delete(k)$: From location $h(k)$, traverse the associated linked list looking for k until you find it or reach the end. If you find it, delete it. Otherwise, return "not there".

DELETE-WITH-CHAINING$(table, k)$

```
 1  linkedlist_node = table[h(k)]
 2  while linkedlist_node.next ≠ NIL
 3          if linkedlist_node.key = k
 4                  linkedlist_node.previous.next = linkedlist_node.next
 5                  return
 6          linkedlist_node = linkedlist_node.next
 7  if linkedlist_node.key = k
 8          linkedlist_node.previous.next = NIL
 9          return
10  return "not there"
```

$Search(k)$: From location $h(k)$, traverse the associated linked list looking for k until you find it or reach the end. If you find it, return a pointer. Otherwise, return "not there".

SEARCH-WITH-CHAINING$(table, k)$

```
1  linkedlist_node = table[h(k)]
2  while linkedlist_node.next ≠ NIL
3          if linkedlist_node.key = k
4                  return &linkedlist_node
5          linkedlist_node = linkedlist_node.next
6  if linkedlist_node.key = k
7          return &linkedlist_node
8  return "not there"
```

(b) [5 points] Prove correctness of your algorithm from part (a). Specifically, prove that the operations modify the data structure as they should, return the right answers, and preserve the property that the structure contains at most one entry for any particular key.
Hint: You might want to add an invariant of the data structure (that is, a statement that is true after any number of operations are performed) saying where a key k must appear, if it is in the structure.

Solution: It is helpful to define two invariants:

Invariant 1: There is at most one entry for any particular key k.

Invariant 2: If there is an entry for key k in the data structure, then it appears somewhere in the linked list starting from location $h(k)$.

$Insert(k)$: If the key is already in the data structure, then Invariant 2 says it must be in the list starting from location $h(k)$, so the search will find it. In this case, the Insert returns "already there", as required. If it's not there, then of course the search won't find it. In this case, the Insert adds an entry for k and returns a pointer to it, as required.

Insert preserves Invariant 1 since it doesn't add a key if the key is already in the structure. Insert preserves Invariant 2 since it puts the new entry into the right linked list.

$Delete(k)$: If the key is in the data structure, the search will find it, by Invariant 2. Then the operation will delete it. By Invariant 1, there is no other copy of k so k is entirely removed from the structure. If the key is not in the data structure, then the search won't find it. In this case, the Delete will return "not there", as required.

Delete preserves both invariants, trivially, since it doesn't add anything.

$Search(k)$: If the key is in the data structure, the search will find it, by Invariant 2, and return the required pointer. If not, it will return "not there", as needed. Search preserves both invariants, trivially, since it doesn't modify anything.

(c) [5 points] Now consider hashing with open addressing, based on a hash function that yields a sequence $h(k, 1), h(k, 2), \ldots, h(k, m)$ for every k, where m is the size of the hash table. Describe (in words and pseudocode) an algorithm to implement the hash table. That is, describe the data structure, its initialization, and the algorithms for $Insert$, $Delete$, and $Search$.

Solution:

Everywhere in this code, the & operator returns the pointer associated with a particular value.

$Insert(k)$: Traverse the sequence (of hash function values) until you encounter k or NIL, in order to see whether the key is already there. If you encounter k, return "already there". If not, then traverse the sequence again from the beginning to the first

NIL or DEL, maintaining a counter of where you are in the sequence. Put the new entry into the first location containing NIL or DEL, and return a pointer to that entry.

INSERT-WITH-OPEN-ADDRESSING($table, k$)

```
1  for i ← 1 to m
2          if table[h(k, i)] = k
3                  return "already there"
4          if table[h(k, i)] = NIL
5                  break
6  for i ← 1 to m
7          if table[h(k, i)] = NIL or DEL
8                  table[h(k, i)] = k
9                  return &table[h(k, i)]
```

$Delete(k)$: Traverse the sequence until you encounter k or NIL. If you first find k, then remove it and replace it with DEL. If you first find NIL, then return "not there".

DELETE-WITH-OPEN-ADDRESSING($table, k$)

```
1  for i ← 1 to m
2          if table[h(k, i)] = k
3                  table[h(k, i)] = DEL
4          if table[h(k, i)] = NIL
5                  return "not there"
```

$Search(k)$: Traverse the sequence until you find k or NIL. If you find k, then return a pointer to it. If you find NIL, then return "not there".

SEARCH-WITH-OPEN-ADDRESSING($table, k$)

```
1  for i ← 1 to m
2          if table[h(k, i)] = k
3                  return &table[h(k, i)]
4          if table[h(k, i)] = NIL
5                  return "not there"
```

(d) [5 points] Prove correctness of your algorithm from part (c), that is, prove that the operations modify the table as they should, return the right answers, and preserve the property that the structure contains at most one entry for any particular key.
Hint: You might want to add an invariant of the data structure saying where a key k must appear, if it is in the structure.

Solution: Define two invariants:

Invariant 1: There is at most one entry for any particular key k.

Invariant 2: If there is an entry for key k in the table, then it appears in some location in the list $h(k,1), h(k,2), \ldots, h(k,m)$, and no preceding location in the list contains NIL.

For all three operations, if the key is already in the table, the initial search will find it, by Invariant 2 and the fact that the operations all search up to the first NIL.

$Insert(k)$: If k is already in the table, the Insert operation finds it in the initial search, as noted above. In that case, the Insert returns "already there", as required. If it's not there, then of course the initial search won't find it. Then the Insert adds an entry for the key and returns a pointer to it, as required.

Insert preserves Invariant 1 since it doesn't add a key if the key is already in the table. It preserves Invariant 2 since it puts the new entry into the correct sequence, and with no intervening NIL entries (since it uses the position of the first NIL or DEL).

$Delete(k)$: If k is in the data structure, the search will find it, as noted above. Then the operation will delete it, by replacing the entry by DEL. By Invariant 1, there is no other copy of k in the table, so k is actually removed from table. If the key is not in the table, then the search won't find it. It will return "not there", as required.

Delete preserves Invariant 1, because it doesn't add anything. It preserves Invariant 2, because it doesn't make any entries NIL, which is the only way Invariant 2 could become false if was true before the operation.

$Search(k)$: If the key is in the table, the search will find it, as noted above, and return the required pointer. If not, of course it won't find it, and will return "not there", as needed. Search preserves both invariants, since it doesn't modify anything.

An alternative invariant that could be used in such a proof is:
Invariant 2': For every key k, the elements of the sequence $h(i,1), h(i,2), \ldots, h(i,m)$ that contain non-NIL entries (that is, keys or DELs) form a prefix of the sequence.

Problem 4-2. [25 points] **Robust Communication Networks**

A communication network, such as the Internet, or perhaps a future network connecting installations on Mars, can be modeled as an undirected graph $G = (V, E)$. Here the vertices V are the machines on the network, and the edge set consists of one edge for each pair of machines that are directly connected. We assume that the edges of G are undirected, that is, if there is a direct connection from machine u to machine v, then there is also a direct connection from machine v to machine u.

It is highly desirable for a communication network graph to be *connected*, so that every machine on the network can communicate, possibly through a series of relays, with any other machine. But networks can change, with some machines failing and other machines being added to the network. It is useful to have a *monitoring algorithm* that collects information about the current network graph (vertices and edges) at designated times, and determines properties related to connectivity.

(a) [4 points] Describe (in words and pseudocode) a monitoring algorithm, which does the following. As input, it is given an undirected graph $G = (V, E)$ representing the current network, in adjacency list format. It should output a Boolean saying whether on not G is connected. Your algorithm should run in time $O(V + E)$.

Solution: Use ordinary DFS, starting from any single source node u. Mark nodes when they are added to the DFS tree. After DFS completes the search from u, do a linear scan of all the vertices to see whether all the nodes are marked. The graph is connected iff all the nodes in the original graph are marked.

To prove this last claim, note that, if the DFS from u marks all nodes, then a path exists between every two nodes, in fact, a path involving only edges of the DFS tree. Conversely, if the graph is connected, then all nodes will be visited during the DFS.

```
DFS-CONNECTED(VertexSet, EdgeSet, root)
1  label root as discovered
2  for v in VertexSet adjacent to root
3          if v not labeled discovered
4                  DFS-CONNECTED(VertexSet, EdgeSet, v)
5  for v in VertexSet
6          if v not labeled discovered
7                  return false
8  return true
```

It's very nice if the network is connected, but even if it is, we might worry that it might become disconnected soon. This is likely if the network graph has *critical vertices*, whose removal would disconnect the graph. That is, u is a critical vertex exactly if there are two other vertices, v and w, for which every connecting path in the graph runs through u (this is equivalent to saying that removing u disconnects the graph, because removing u leaves no path from v to w.)

In the next three parts of the problem, you will develop an algorithm to find all the critical vertices in a connected network graph.

(b) [4 points] Suppose we are given a network graph $G = (V, E)$ that we know is connected, and suppose we are given any DFS tree T for G. Under what conditions is the root of T a critical vertex of G? Prove your answer.

Solution: The root of the DFS tree is critical iff it has more than one child in the DFS tree. We prove both directions of this statement.

First, we show that, if the root is critical, then it must have more than one child. Suppose for contradiction that it has 0 or 1 children. If it has 0 children, then the graph consists of a single vertex, and removing it would leave no nodes, so cannot disconnect any nodes. If the root has exactly 1 child in the DFS tree, then removing it leaves a single connected component containing all the remaining vertices, so no other nodes are disconnected. Either way, we have a contradiction, which means that the root must have more than one child.

Second, we show that, if the root has more than one child, then it is critical. Suppose the root u has two distinct children in the DFS tree, v and w. For the sake of contradiction, suppose that removing u does not disconnect the remaining subgraph. In particular, there must exist a path from v to w that does not go through u. But in this case, whichever of v and w was visited first in the DFS would be an ancestor of the other in the DFS tree. For example, if v was visited first, w could not be a child of u, the root, because it would have been visited through v since a path exists. This is a contradiction.

Alternatively, we can say that, if v and w are connected by a path that does not go through u, then there must exist some cross edge between a descendant of v and some descendant of w. But DFS trees for undirected graphs do not contain cross edges, contradiction.

(c) [6 points] Suppose we are given a network graph $G = (V, E)$ that we know is connected, and suppose we are given any DFS tree T for G. Under what conditions is a particular *non-root node* u of T a critical vertex of G? Prove your answer.
Hint: Consider back-edges between proper descendants of u and proper ancestors of u.

Solution: We claim that non-root node u is critical iff it has some child v such that for every pair (x, y), where x is a descendant of v and y is a proper ancestor of u, there is no back-edge from x to y.

First, we show that, if u has such a child v, then u is critical. If u has such a child v, then removing u disconnects v from u's parent. To see this, note that there are no back edges from v or its descendants to u's ancestors, and there are no cross edges (because this is an undirected DFS tree). Thus, there can be no path from v to u's parent that does not go through u.

Second, we show that, if u is critical, then it has such a child v. Suppose for contradiction that all children of u fail to have this property—that is, for every child v of u, some back-edge exists between a descendant of v and a proper ancestor of u. Then removing u cannot disconnect the graph, because all children of u as well as their descendants are still connected to the rest of the graph by these back-edges. Thus, u is not critical, contradiction.

(d) [7 points] Describe (in words and pseudocode) a new monitoring algorithm, which does the following. As input, it is given a *connected* undirected graph $G = (V, E)$ representing the current network, in adjacency list format. It should output a list of all the critical vertices of the graph. Your algorithm should run in time $O(V + E)$.

Solution: Run DFS on the graph, producing a tree T. We determine whether the root satisfies the condition from part (b) and determine whether each non-root node satisfies the condition from part (c). For the root, we need only check the number of children it has, and add it to the output list if it has at least two.

For non-root nodes, implementing the test from part (c) is more complicated. For each vertex u, we keep track of two attributes, $disc(u)$ and $min(u)$. $disc(u)$ is the discovery time of the vertex, as defined in lecture, that is, the time at which it was visited in the DFS traversal, where the root is 1, the next vertex visited is 2, etc. $min(u)$ is the minimum value of $disc$ among the set consisting of u and all ancestors y of u for which there exists a back-edge from a descendant of u to y. That is, if $min(u) \neq disc(u)$, then it is the discovery time of the first-discovered ancestor of u that can be reached from a descendant of u by following one back edge.

Now, part (c) says that non-root node u is critical if and only if it has some child v such that there is no back edge from any descendant of v to a proper ancestor of u. That is the same as saying that u has some child v such that $min(v) \geq disc(u)$. To see this equivalence, note that proper ancestors of u are discovered strictly before u, so they have $disc$ values that are smaller than $disc(u)$. So $min(v) < disc(u)$ is another way of saying that there is a back edge from a descendant of v to a proper ancestor of u.

Now we analyze the time complexity. First, we claim that we can compute $disc(u)$ and $min(u)$ for all u in the graph in time $O(V + E)$. We can compute the $disc$ values easily during our DFS traversal; computing the min values takes some more work.

For example, after building the DFS tree, we can compute $min(u)$, working recursively on the tree. For each leaf u, all incident edges except the one connecting u to its parent are back edges. For each such back edge, we determine the $disc$ value for the node at the other end. We compute $min(u)$ as the minimum of all these $disc$ values and $disc(u)$.

For each internal node u, all non-parent incident edges are either tree edges connecting u to its children, or back edges connecting u to an ancestor. For each child v or u, we determine $min(v)$ when the recursive call for v returns. For the back edges from u,

we determine the $disc$ values as we did for the leaves. Finally, we compute $min(u)$ as the minimum of: the min values for all its children, the $disc$ values encountered by the back-edges from u, and $disc(u)$. We can do this without asymptotically increasing the cost of DFS, because we only use each edge once in this phase.

Once we have computed all the $disc$ and min values, we can traverse the tree again, looking for nodes u that have some child v such that $min(v) \geq disc(u)$. Any such node u is critical, and we output it. This traversal can be done in time $O(V + E)$ (actually $O(E)$ since the tree is connected). Thus, the entire algorithm for computing the critical vertices runs in $O(V + E)$, as desired.

```
AUGMENTED-DFS(VertexSet, EdgeSet, root)
 1  label root as discovered
 2  root.setDisc(counter) // counter is a global variable we initialize to 0
 3  root.setMin(root.getDisc())
 4  for v in VertexSet adjacent to root
 5          if v not labeled discovered
 6                  child = AUGMENTED-DFS(VertexSet, EdgeSet, v)
 7                  root.setMin(min(root.getMin(), child.getMin()))
 8                  root.addChild(child)
 9          else
10                  root.setMin(min(root.getMin(), v.getDisc()))
11  return root
```

```
CRITICAL-VERTICES(VertexSet, EdgeSet, root)
1  CriticalVertices = []
2  if root.getChildren().size() > 1
3          CriticalVertices.append(root)
4  for u in VertexSet not equal to root
5          for v in u.getChildren()
6                  if v.getMin() ≥ u.getDisc()
7                          CriticalVertices.append(u)
8                          break
9  return CriticalVertices
```

Now suppose the network is connected, and moreover, it has no critical vertices. Thus, there is no single vertex whose removal would disconnect the network, so the network is robust to a single machine failure. But we might still not be satisfied: What if the removal of *two* vertices could disconnect the graph? Define $\{u, v\}$ to be a *critical pair of vertices* if there are two other vertices, w and x, for which every connecting path runs through at least one of u and v.

(e) [4 points] Describe (in words and pseudocode) yet another monitoring algorithm, which does the following. As input, it is given an undirected graph $G = (V, E)$ representing the current network, in adjacency list format. This time, not only do

we know that G is connected, but we also know that it has no critical vertices. The algorithm should output a list of all the critical pairs of vertices. It should run in time $O(V(V+E))$.

Solution: We can simply consider each node u of the graph in turn. For each u, remove u from the graph. The remaining graph is still connected by assumption (it's given that there are no critical vertices). Then apply part (d) - any critical vertices on the remaining graph, coupled with u, will form a critical pair. This takes $O(V(V+E))$.

```
CRITICAL-PAIRS(VertexSet, EdgeSet)
1  critical_pairs = []
2  for v in VertexSet
3          TempVertexSet = VertexSet − v
4          root = AUGMENTED-DFS(TempVertexSet, EdgeSet, TempVertexSet[0])
5          TempCriticalVertices = CRITICAL-VERTICES(TempVertexSet, EdgeSet, root)
6          for u in TempCriticalVertices
7                  critical_pairs.append((u, v))
8  return critical_pairs
```

Problem 4-3. [15 points] **Lemon-Meringue Pie**

During Spring Break, Emilie's grandmother taught her how make lemon-meringue pie (http://allrecipes.com/recipe/15093/grandmas-lemon-meringue-pie/). Upon returning, Emilie can remember all the steps she needs to perform, but unfortunately, she cannot recall the precise order in which she is supposed to do them. However, being a logical person, she figures that the precise total order cannot be important, but certain steps must precede other steps.

Emilie recalls that the recipe involves the following steps:

1. Spread meringue on top of filling.
2. Cook filling until it thickens.
3. Combine flour, sugar, and cornstarch.
4. Heat oven.
5. Bake pie shell.
6. Whip sugar and egg whites to form a meringue.
7. Pour filling into baked pie shell.
8. Make pie shell.
9. Add diluted lemon juice to dry ingredients.
10. Bake pie until golden brown.
11. Add butter and egg yolks to filling.
12. Cook filling until it boils.
13. Dilute lemon juice.

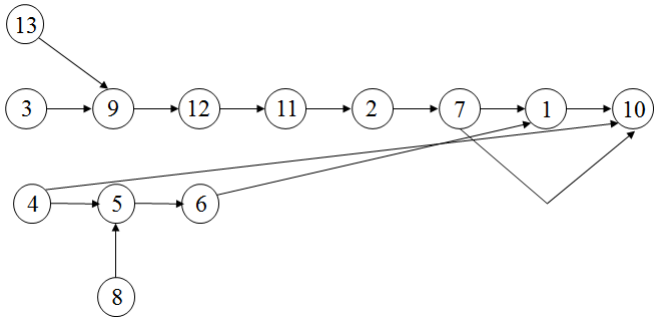
And she can deduce the following common-sense constraints on the order in which to perform the steps:

- 4 must precede 5 and 10
- 6 must precede 1
- 1 must precede 10
- 13 and 3 must precede 9
- 9 must precede 12
- 11 must precede 2
- 12 must precede 11
- 2 must precede 7
- 7 must precede 1 and 10
- 8 must precede 5

- 5 must precede 6 and 7

(a) [3 points] Draw a directed graph with states corresponding to the 13 steps of the recipe and a directed edge from any step to any other step that must follow it according to Emilie's constraints. A topological sort of this graph will give a feasible order for performing the steps.

Solution:
We can interpret the constraints as a digraph:



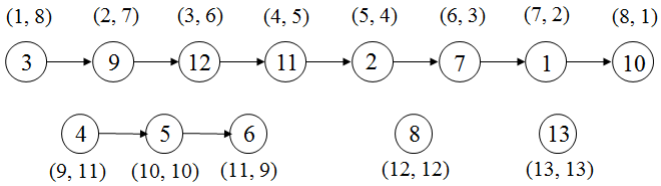
(b) [3 points] Emulate DFS on the graph and draw a picture of the resulting forest, plus a list of the order of events in which you discover and finish each node. Start with the lowest-numbered node that has no precedessors, and from then on, when you must make a choice, choose the lowest-numbered unvisited node. Identify edges as tree edges, forward edges, or cross edges.

Solution: The following is a list of the order of events in which nodes are discovered and finished, grouped by the tree in the forest to which they belong (conveniently, all trees are paths).

- Discover 3, 9, 12, 11, 2, 7, 1, 10
- Finish 10, 1, 7, 2, 11, 12, 9, 3
- Discover 4, 5, 6
- Finish 6, 5, 4
- Discover 8
- Finish 8
- Discover 13
- Finish 13

The only forward edge is (7, 10).
The cross edges are (4, 10), (5, 7), (6, 1), (8, 5), and (13, 9).
All other edges are tree edges. There are 15 edges, as desired.

Here is a diagram of the resulting forest. Each node is labeled with a pair describing its (discovery time, finish time).



(c) [3 points] Demonstrate that the order of *discover* events in your list in part (b) does not yield a correct order for the steps, that is, performing the steps in this order does not produce a lemon-meringue pie.

Solution: For example, we discover node 10 before we discover node 4, whereas the rules require that 4 precedes 10. That is, we cannot bake the pie before we heat the oven.

(d) [3 points] Now consider the *reverse* of the order of *finish* events (topological sort). List the steps. Does this satisfy all the ordering constraints? Would this correctly prepare the pie?

Solution:

The reverse order of events is:

1. Dilute lemon juice (Step 13)
2. Make pie shell (Step 8)
3. Heat the oven (Step 4)
4. Bake pie shell (Step 5)
5. Make the meringue (Step 6)
6. Combine dry ingredients (Step 3)
7. Add diluted lemon juice to dry ingredients (Step 9)
8. Cook filling until it boils (Step 12)
9. Add butter and egg yolks (Step 11)
10. Cook filling until it thickens (Step 2)
11. Pour filling into baked pie shell (Step 7)
12. Spread meringue on top of filling (Step 1)
13. Bake pie (Step 10)

This does satisfy all the ordering constraints. Based on common sense, this should produce a fine pie. You could try making it.

(e) [3 points] The key to why this works is that, in performing the DFS of an acyclic directed graph, there are no situations where there is an edge in the graph from a node u to another node v, but u finishes before v finishes. Explain why no such situation arises, in your own words.

Solution: Suppose for contradiction that we have an edge from u to v but u finishes before v does. In our reverse ordering, this says that we would perform v before u, which is not allowed.

Since u finishes before v does, either u is a descendant of v or u is to the left (traversed first) of v, in the same tree or a previous tree.

If u is a descendant of v then we have a back edge from u to v and a path from v to u. This yields a cycle, which contradicts the assumption that the graph is acyclic. If u is to the left of v, then u finishes before v is discovered. But u would not finish before exploring its edge to v, which had not been discovered at that point. Thus this is also a contradiction.

So, in either case we get a contradiction. That means that we cannot have an edge from u to v, while v finishes before v does.

Part B

Problem 4-4. [40 points] **Model-Checking**

Model-checking is a method whereby an algorithm or system is modeled abstractly, as a rooted directed graph, where the nodes of the graph correspond to the possible *states* of the algorithm or system, and the edges correspond to possible transitions from one state to another. The root node corresponds to the *start state*, s_0. The models are typically *nondeterministic*, in that there might be many transitions from the same state s to other states, represented by many edges leaving the node labeled s. Model-checkers use Breadth First Search (BFS) and other searching techniques to explore the states of the graph, in order to determine whether or not all the states that are *reachable* from the start state satisfy some desirable property P, that is, if P is an *invariant* of the state machine represented by the graph. Or, one might use a model-checker to determine whether a particular state t is reachable from the start state s_0. The inventors of model-checking, Ed Clarke, Allen Emerson, and Joseph Sifakis, won the Turing award in 2007 for their work.

In this problem you will build a model-checker using BFS and use it to check some properties of some simple systems. All the systems will have states that are triples of integers in a bounded range $0, \ldots, k$.

(a) [16 points] Write a Python program that expects as input a representation of a system, consisting of:

- k, a bound on the integers used in the state triples.
- A start state s_0.
- A list of transitions of the form (s, t), where s and t are states.

Your algorithm should output a list of all the reachable states of the system. Each state s should be accompanied by the length of a shortest path from the start state s_0 to s, and the predecessor of s on some shortest path from s_0. These will be represented by a list of 3-tuples in the code, where the three elements are the state s, the length of the shortest path from s_0 to s, and the predecessor. The outputs should appear in order of path lengths.

Solution: First transform the given representation into a standard adjacency list graph representation.

We could do this by inserting all of the transitions into a dictionary. The key would be the start state, which would map to a list of states that the start state can transition to. Then, for each state, we would have a mapping from the state to a list of possible next states, which is exactly an adjacency list.

Another approach is to use radix sort, based on the three components of the first states in the pairs (s, t) in the transition list. This makes all the pairs (s, t) having the same s consecutive. From there, a simple linear scan can put them into adjacency-list form.

Now, using the adjacency list representation, run BFS on the graph, starting from node s_0. This should return a list of all the nodes in the graph that are reachable from s_0 as well as the length of the shortest path from s_0 to each node and the needed predecessor.

We can analyze the runtime of this algorithm. First, say that there are n transitions. Converting a list of transitions using a dictionary takes $O(n)$ time, since all that is needed is to loop over every element once and insert each element into the dictionary. Similarly, a radix sort would take $O(n)$ time. Running BFS would take $O(n + k^3)$ time, since there n edges in the graph and k^3 possible states. Therefore, the algorithm takes $O(n) + O(n + k^3) = O(n + k^3)$ time.

(b) [8 points] Consider a simple state machine with some initial state s_0 and transitions that satisfy the following rules. Each rule denotes a set of transitions, one for each triple (a, b, c) with components in the required range $\{0, \ldots, k\}$ and for which the resulting triple also has its components in the required range.

1. $(a, b, c) \to (a + 1, b + 1, c + 1)$.
2. $(a, b, c) \to (a - 1, b - 1, c - 1)$.
3. $(a, b, c) \to (a + 1, b, c)$.
4. $(a, b, c) \to (a - 1, b, c)$.

Write a program that takes as inputs the number k, an initial state s_0, and a target state t, and outputs a shortest path from s_0 to t using the given rules. This should be a list of states starting with s_0 and ending with t. If there is no path, then return a value of False.

Solution: First, convert the list of given rules into transitions for the program. This is a set of straightforward loops for values of a, b, and c from 0 to k. The program should check that the transitions only use values from 0 to k, i.e. not -1 or k+1.

With the transitions, then use the program from part (a) to compute the list of reachable states from s_0. To actually extract a path from s_0 to t, if it exists, you must reconstruct the path from the previous state associated with each state. To do this, loop backwards through the list of reachable states. This order will be decreasing in path length, and the previous state is always earlier in the list of reachable states, so we are guaranteed to reconstruct a path to t if it does exist.

We can now show runtime. Generating a list of transitions takes $O(k^3)$ time, since there are $O(k^3)$ transitions. The program from part (a) takes $O(n+k^3)$ time, as shown in part (a)'s solution. Looping through the list of reachable states is $O(k^3)$ time, since there may be $O(k^3)$ reachable states. The total runtime is then $O(k^3) + O(n + k^3) + O(k^3) = O(n + k^3)$.

In the remaining examples, we will consider simple state machines that represent plausible two-process *Mutual Exclusion* algorithms, which two concurrent processes can use to coordinate their requests to obtain access to an unsharable resource. The states of these algorithms consist of triples of integers in $\{0, 1, 2, 3\}$, that is, $k = 3$. The first two components of the triples represent the level

of progress of processes 1 and 2, respectively, with level 0 meaning that the process isn't involved at all, 1 and 2 indicating that it is trying to get the resource, and 3 meaning that it has the resource. The third component of the triples represents some extra data used in the algorithm. A mutual exclusion algorithm is supposed to guarantee that it is never the case that both processes have the resource at the same time; that is, we want to avoid states of the form $(3, 3, c)$.

(c) [8 points] **Mutual Exclusion 1**

This is a variant of Gary Peterson's two-process mutual exclusion algorithm. In this algorithm, the third component of the triples remembers the index of the last process that started trying to obtain the resource, that is, that raised its level from 0 to 1. The initial state is the triple $(0, 0, 1)$, which means that neither process is involved and process 1 is the latest one that tried to obtain the resource (this is just an arbitrary default).

The behavior of the algorithm is described by the following four types of rules:

1. $(0, b, c) \to (1, b, 1)$, and $(a, 0, c) \to (a, 1, 2)$.
   That is, if a process is at level 0, it can advance to level 1, and record its index in the third component. This means that this process was the latest one that tried to obtain the resource.
2. $(1, 0, c) \to (3, 0, c)$, and $(0, 1, c) \to (0, 3, c)$.
   If one process is at level 1 and the other process is at level 0, then the process at level 1 can obtain the resource.
3. $(1, b, 2) \to (3, b, 2)$, and $(a, 1, 1) \to (a, 3, 1)$.
   If one process, say i, is at level 1 and the other process was the latest to try to obtain the resource, then process i can obtain the resource.
4. $(3, b, c) \to (0, b, c)$ and $(a, 3, c) \to (a, 0, c)$.
   Any process that has the resource can give it up and return to being uninvolved.

Write a program that determines whether or not the algorithm satisfies the mutual exclusion property, that is, whether or not some state of the form $(3, 3, c)$ is reachable from the initial state of $(0, 0, 1)$. The program should output False if the algorithm satisfies the mutual exclusion property. Otherwise, it should output a minimum-length counterexample, that is, a shortest execution that leads to a violation. This should be a list of states starting with s_0 and ending with t.

Solution: This solution is very similar to the solution from part (b). Convert these rules into a list of transitions. This can be done either with a loop like before or by just writing them all manually. Then, use the program from (a) to generate a list of reachable states, then loop backwards through the list of reachable states to generate a path from the initial state, $(0, 0, 1)$, to any of the end states, $(3, 3, c)$, for all values of c, if any such path exists. Since this should be a correct ME algorithm, the answer should be that no such triples appear.

(d) [8 points] **Mutual Exclusion 2**

This algorithm is a variant of Michael Fischer's two-process mutual exclusion algorithm. In this algorithm, the third component remembers the index of a process that has advanced to a certain point in securing the resource. The initial state is the triple $(0, 0, 0)$.

The behavior of the algorithm is described by the following rules:

1. $(0, b, 0) \to (1, b, 0)$, and $(a, 0, 0) \to (a, 1, 0)$.
   If a process is at level 0 and the third component is 0, then the process can advance to level 1.
2. $(1, b, c) \to (2, b, 1)$, and $(a, 1, c) \to (a, 2, 2)$.
   If a process is at level 1, then it can set the third component to its index and advance to level 2.
3. $(2, b, 1) \to (3, b, 1)$, and $(a, 2, 2) \to (a, 3, 2)$.
   If a process is at level 2 and sees the third component (still) equal to its index, then it can obtain the resource.
4. $(2, b, c) \to (0, b, c)$, for $c \neq 1$, and $(a, 2, c) \to (a, 0, c)$, for $c \neq 2$.
   If a process is at level 2 and sees the third component unequal to its index, then it can drop back to level 0.
5. $(3, b, c) \to (0, b, 0)$ and $(a, 3, c) \to (a, 0, 0)$.
   Any process that has the resource can give it up and return to being uninvolved. It also sets the third component to 0.

Write a program that determines whether or not the algorithm satisfies the mutual exclusion property, that is, whether or not some state of the form $(3, 3, c)$ is reachable from the initial state of $(0, 0, 0)$. The program should output False if the algorithm satisfies the mutual exclusion property. Otherwise, it should output a minimum-length counterexample, that is, a shortest execution that leads to a violation. This should be a list of states starting with s_0 and ending with t.

Solution: This solution is almost exactly the same as part (c). Convert the rules into a set of transitions, and generate a list of reachable states from the initial state $(0, 0, 0)$ to $(3, 3, c)$ for all values of c. Since this is not a correct ME algorithm, the answer should be that there is some path. Use the algorithm from part (b) to generate a path from $(0, 0, 0)$ to $(3, 3, c)$.