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 $Search$.

(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.

(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 $Search$.

(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.

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 $O(V + E)$.

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.

(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.

(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 $O(V + E)$.

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 w and x.

(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 $O(V(V + E))$.

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.

(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.

(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.

(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?

(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.

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.

(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.

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.

(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.