Project 3: Oracle

Introduction

In this assignment, you will develop an *oracle* to test (possibly broken) solutions to the *Stable Matching Problem*. This assignment is different in that:

- Your purpose is not to solve the problem, but to test whether a solution given to you is correct. That is, all code you write will be tests and supporting code for tests.
- There might be no unique solution. Thus, you will need to test *properties* of the given solution, and make sure you are considering enough properties so anything that satisfies all of them (i.e., passes all your tests) is correct.

The Stable Matching Problem

You can learn more about it from Wikipedia, but a summary of the problem is as follows:

Assume you have two sets, A and B, where each set has *n* members. We need to **match** every member in A with **exactly one** member of B and vice versa.

Preferences: Every member of each set ranks all members of the other set by preference, in an array (the first preference at index 0, the least preferred partner at index n-1).

For this assignment, we match **companies** with **candidates**. Each company has a ranking of all candidates, in order of preference. Each candidate has a ranking of all companies. A programmer has implemented a program that generates a matching: n pairs, each with a unique company and candidate.

Stable matching. The program's generated "hires" are stable if there do **not** exist two matched pairs, where a company from one pair **and** a candidate from the other pair **both** prefer each other to their current match (the preferences of their current partners don't matter).

There are many problems of this nature. Consider assigning TAs to classes; matching residents with hospitals; pairing students for homework; and much more. Some of these problems are variation on the theme (maybe the companies don't rank every candidate, or are allowed to give some of them the same rank; maybe you only have partial information for making the assignment; etc.). Ultimately, however, this problem in its many guises has wide application.

Assignment Overview

Writing code that you think is right is not enough for anyone to be confident that your software is correct. However, almost no software complex enough to be useful can be proved correct by hand or automatically in a reasonable amount of time. A computer scientist's solution to this problem is automated testing. Your job in this assignment is to build an automated testing **oracle** for a hypothetical solution to the stable matching problem.

Your oracle's job is to generate and feed test inputs to a given solution and test the correctness of the output. In the past, you did this by comparing the output to a precomputed right answer. This assumes two things: that there is only one right answer, and that it is easy for you to find it. In the real world, either of these can be false. (How do you know what the right answer to an arbitrary instance of the problem is if the original problem was to write a program to find it?)

We will generate both correct and incorrect solutions of the stable matching problem to help you write and test your oracle. You submit your oracle, and we will evaluate its performance in classifying various implementations as correct or incorrect.

Input-Output Specification

The preferences of a candidate are represented as an array of *N* distinct numbers from 0 to *N* - 1. For example, if a candidate's preferences are the array [2, 0, 1], that means that that candidate's most preferred company is company 2 and least preferred company is company 1. A company's preferences are represented in the same way, but ranking candidates instead. All these are grouped into one array of company preferences, and one array of candidate preferences.

We have added the following functions to Ocelot that you can use in this assignment:

```
wheat1(companies: number[][], candidates: number[][]): Hire[]
chaff1(companies: number[][], candidates: number[][]): Hire[]
```

Type Hire denotes an object with two fields: `{ company: number, candidate: number }`

Both functions `wheat1` and `chaff1` take as arguments preference arrays and return solutions to the stable matching problem. However, **`chaff1` has bugs**, whereas `wheat1` gives a correct solution. The number *N* of companies and candidates must be the same. `companies[i]` is the preference array of company `i` (an array of N candidate numbers, in preference order) and `candidates[j]` is the preference array of candidate `j`.

The result of both `wheat1` and `chaff1` are an array of N `Hire` objects, where each object matches a company with a candidate. The returned array of hires is not in any particular order.

Programming Task: Part 1 (due Tue, 9 March 2021)

1. **Write the following function:**

```
generateInput(n: number): number[][]
```

This function should produce an n by n array of preferences for companies or candidates. This will be used as input for testing a given solution. Your function should generate random values in order to test a given solution with a broad spectrum of input.

Generating Random Numbers. This can be done using `Math.random()` and `Math.floor()`.

```
// Returns a random int i where min <= i < max
function randomInt(min, max) {
 return Math.floor(Math.random() * (max - min)) + min;
}
```

2. **Write the following function:**

```
oracle(f: (companies: number[][], candidates: number[][]) => Hire[]): void
```

This function should test an implementation of stable matching. It will be called with a function (like `wheat` or `chaff`) that generates a matching (stable or not) from the given preference arrays.

You have no access to the implementation of `f`, so you will test its outputs on many inputs. There are numerous ways for an algorithm to return an incorrect solution. You will have to figure out all properties that make a solution (in)correct and implement tests for all of them. However, even if an implementation is incorrect, it may sometimes produce a correct solution. This is fine, your only requirement is to correctly classify each solution as right or wrong.

A template for the `oracle` function is given below. To do well, you should carefully consider all the different ways in which the output could be either invalid for the original problem statement. You may assume the output is of the right type, `Hire[]`, but nothing else.

```
function oracle(f) {
 let numTests = 20;  // Change this to some reasonably large value
 for (let i = 0; i < numTests; ++i) {
   let n = 6;  // Change this to some reasonable size
   let companies = generateInput(n);
   let candidates = generateInput(n);
   let hires = f(companies, candidates);
   test('Hires length is correct', function() {
     assert(companies.length === hires.length);
   }); // Write more tests like this one
 }
}
```

You can then call `oracle` on the test inputs with either `oracle(wheat1)` or `oracle(chaff1)`. You can press the "Test" button to run the tests in `oracle`. Note that this `oracle` function does not return anything. The goal of `oracle` is to have all its tests successfully pass when given a correct implementation and at least one test fail for wrong outputs of a faulty implementation.

Programming Task: Part 2 (due Tue, 16 March 2021)

3. Now test whether the stable matching algorithm indeed performs the specified steps.

Algorithm: We assume the following non-standard variant: At any step, any *unmatched* company or candidate may propose. Every party always proposes to the *next* potential partner on their preference list, starting with the top choice. Proposals are not repeated. Any unmatched party that receives a proposal accepts unconditionally. If the receiving party is already matched, but they receive a better offer (higher in their preference list), they accept, and their current partner becomes unmatched; otherwise, the offer is rejected. The algorithm ends when all parties are either matched or have made offers to the entire preference list. The algorithm is underspecified/nondeterministic: it doesn't state whether a company or a candidate proposes at a given step, nor which one does, as long as the given rules are observed.

For this problem, consider the following additional types:

```
type Offer = { from: number, to: number, fromCo: boolean }
type Run = { trace: Offer[], out: Hire[] }
```

An **offer** is a triple with two numbers from 0 to n-1 identifying the proposer and the recipient, and a boolean `fromCo`; if `fromCo` is true, the proposer is a company, otherwise a candidate.
A **run** is a sequence of offers, together with an outcome, which is a **matching** (like at point 2).

Write a function

```
runOracle(f: (companies: number[][], candidates: number[][]) => Run): void
```

This function should test the provided implementation of stable matching. It should check that **1)** the offer sequence in the `trace` is **valid**, made according to the given algorithm, starting with all parties unmatched. The trace need not be a complete algorithm run, it may stop at any point. **2)** the produced matching (`out`) is indeed the **result of the offers** in the trace.

You should **not** test whether the resulting matching is *complete* **nor** *stable*. You need to check the constraints at 1) and 2) above. To do this, you might need to "simulate" the effects of the offers in the trace, keeping track of the matchings at every step.

For instance, `{trace: [{from:1, to:0, fromCo:true}, {from:2, to:0, fromCo:true}], {out: [{company:2, candidate:0}]}` is a valid run, assuming candidate 0 is the top preference for both companies 1 and 2, but prefers company 2 to company 1.

Your oracle function should have the same structure as the one at point 2, generating random inputs (preference lists), and defining tests for correct runs and outcomes.

You should call the oracle on test inputs with the following lines:

```
const oracleLib = require('oracle');
runOracle(oracleLib.traceWheat1);
runOracle(oracleLib.traceChaff1);
```

Additional Example

Assume that we have candidates Alice, Bob, and Charles, and companies Acme, Better Mouse Trap, and Cars R Us. The candidate preference matrix could look like this:

Alice	0 (Acme)	1 (BMT)	2 (Cars R Us)
Bob	0 (Acme)	2 (Cars R Us)	1 (BMT)
Charles	2 (Cars R Us)	1 (BMT)	0 (Acme)

This means that Alice's preferences are Acme, Better Mouse Trap, and Cars R Us, in that order. Bob's preferences are Acme, Cars R Us, and Better Mouse Trap, in that order. Charles' preferences are Cars R Us, Better Mouse Trap, and Acme, in that order.

The company preference matrix could look like this:

Acme	0 (Alice)	2 (Charles)	1 (Bob)
Better Mouse Trap	0 (Alice)	1 (Bob)	2 (Charles)
Cars R Us	0 (Alice)	2 (Charles)	1(Bob)

This means that Acme's preferences are Alice, Charles, and Bob, in that order. Better Mouse Trap's preferences are Alice Bob, and Charles, in that order. Cars R Us preferences are Alice, Charles, and Bob, in that order.

A possible trace could look like this: [{from:1, to:1, fromCo:false}]. This means that Bob makes an offer to Better Mouse Trap. **This offer is invalid**, since Bob's top choice is Acme, and Bob did not previously offer to Acme in this run).

Another possible trace could be [{from:2, to:0, fromCo:true}]. This means that Cars R Us makes an offer to Alice. This is a valid offer, since Cars R Us' top choice is Alice, and Cars R Us is unmatched. Alice's top choice is Acme, but being unmatched, she has to accept the offer. The result is that Alice and Cars R Us are now marked as matched. If later in the run Alice receives an offer from Acme, Alice accepts and is matched to Acme, and Cars R Us becomes unmatched.

Here is a longer trace: [{from:2, to:0, fromCo:true}, {from:0, to:0, fromCo:true}, {from:2, to:2, fromCo:true}]. This is a valid run. A valid resulting Hire array would be [{ company: 0, candidate: 0 }, { company: 2, candidate, 2 }], which could also be depicted by the matrix below.

	Alice	Bob	Charles
Acme	X		
Better Mouse Trap			
Cars R Us			X