GeoMatch Technical Assessment

For the following problems, please feel free to use whatever language you feel most comfortable with. In the attached folder, you will find a directory called `data` which contains various input files for the problems below. The problems below will ask you to write programs that generate CSVs. Your final submission should be your **zipped** working directory. The working directory should contain the following directories / files:

- The CSV output to the problems below must be placed in a **flat / not nested** `output` directory. The problems below describe output file naming.
- If your chosen language's conventions allow, code should be in a directory called `src`
- Each problem has a few followup questions, which are provided in markdown format in a file called `followups.md`. Add responses to these questions in the markdown file directly. Your followup answers shouldn't contain code, and you should not edit any of your source code based on the follow up questions. Your answers shouldn't be more than 250 words (and they may be less).

The final directory structure should look like:

```
$ ls geomatch-assessment
data
src
output
followups.md
```

And your submission might be generated like:

```
$ zip -r [any-4-digit-id]-solutions.zip geomatch-assessment

# NOTE: Please remove any package directories (like .venv) from the final folder
```

You are not required to provide instructions on how to run the code, but please comment and document your code as you think appropriate.

You may depend on any external packages you would like. You may be asked to modify and discuss your solution in a followup interview.

For these problems, clear and maintainable code is preferable to highly optimized code.

We recommend reading problems all first, as they build on each other.

Introduction

At GeoMatch, our goal is to use machine learning to improve refugee outcomes by recommending host cities (aka `locations`) for each group of refugees. Refugees arriving in a country are assigned to locations by a refugee agency in the host country, with a placement officer determining the most suitable location for each group of refugees (a group of individual refugees is referred to as a "`case`").

In this context, the placement officer would be the user of GeoMatch, and GeoMatch would be tasked with providing a ranked list of locations for a given case.

Problem 1: Determining Case Groups

One common challenge with refugee case management is that members of the same family might be registered as separate cases, and each case might travel to their host countries separately.

We refer to this link between two cases as a `”cross reference”`.

An example list of cross references is provided in `data/p1_cross_references.csv`. Each row in this CSV contains a cross reference, and each will have the following (required) columns:

- **`from_case_id`** - This is the case ID of the first case in the cross reference
- **`to_case_id`** - This is the case ID of the second case in the cross reference
- **`from_case_size`** - This is the number of individuals in the first case
- **`to_case_size`** - This is the number of individuals in the second case
- **`from_to_relationship`** - This is the relationship between the representative individual of the first case, to the representative individual of the second case. It has one of the following possible values: `[spouse, parent, child, cousin]`
    - If (`from_case_id, to_case_id, from_to_relationship`) == (`1, 2, parent`), then case 2 is called the parent of case 1.

Here are the characteristics of the data:

- `case_ids` fall in the range from [0,1000] (inclusive)
- Groups may have 1 or more cases
- You can assume there will be no more than 10,000 cross references in the input
- Cross references are bidirectional / undirected in practice. If a cross reference exists with `from_case_id -> to_case_id` as `1 -> 2` and/or `2 -> 1` that means that case 1 and case 2 are in the same group.

        - In this example, the edges `1 -> 2`, `2 -> 1`, or both may appear in the list of cross references. In any of these situations, it simply means that case 1 and case 2 are in the same group.
    - If both directions are represented as a cross reference, you can assume that the case size and relationship fields are consistent in both directions.
        - In other words, if you see a cross reference from `1 -> 2` with a relationship of `parent`; if a `2 -> 1` reference exists, the relationship will be `child`.

Your goal is to identify each grouping of cases. `case_x` and `case_y` are said to be in a group if `case_x` is connected to `case_y` through **any number of non-cousin cross references**. For example, if `case_x` is a parent of `case_z`, and `case_z` is a parent of `case_y`, then they would all be in the same group (even if no cross references connected X and Y). This further means that cousins should not be in a group together, unless other non-cousin cross references dictate (e.g. in the silly scenario where `case_x` and `case_z` are cousins, they would still be in the same group).

The output should be in JSON format and written to `output/p1_case_groups.json.` This output is a JSON list with one entry for each group of cases, with a new variable called group_id. **group_id should be equal to the lowest case_id in the group.** Both groups and their cases may be written in any order.

Here is an example `p1_case_groups.json`:

```
[
{
  group_id: 2,
  cases: [
    {
      case_id: 3,
      num_individuals: 2
    },
    {
      case_id: 2,
      num_individuals: 2
    },
  ]
},
{
  group_id: 1,
  cases: [
    {
      case_id: 1,
      num_individuals: 2
    }
  ]
},
]
```

Problem 2: Updating Case Groups

You have successfully detected case groups based on the initial data! However, case groups are often affected by unobserved factors that we don't get in the data. In order to account for this, we let GeoMatch users manually modify case groups using their on-the-ground information.

In this problem, you will be creating an "API" to support this use case. The API will have three operations: `/link`, `/unlink`, and `/get-group`.

You are not required to set up a working web server for this problem (but you may if you prefer). Any means of transforming the provided JSON requests into a list of JSON responses will serve as a mock "API" (e.g. simple file IO).

You may also use any datastore of your choosing. It could be an in-memory store that you develop yourself, a SQL database, or a flat file.

The datastore should be seeded with your output from problem 1.

For each request in the `data/p2_requests.json` list, create a response JSON object and add it to a JSON list. This list should be written to `output/p2_responses.json`. The responses should be ordered to match the order of the requests. **You may assume these requests arrive sequentially and will be processed by a single thread.**

A brief example `p2_responses.json` for 2 input requests:

```
[
{
  status: 400
},
{
  status: 200,
  new_group_id: 3
}
]
```

All of the operations' responses have a `status` field that can either be `200` (success) or `400` (bad request). For bad requests, no other response fields should be set, and no database state should be mutated.

/get-group

This returns information about each case in a given group ID.

It will return `400` if the group ID does not exist.

Example request body:

```
{
  group_id: 2
}
```

Example success response:

```
{
  status: 200,
  group_id: 2,
  cases: [
    {
      case_id: 3,
      num_individuals: 2
    },
    {
      case_id: 2,
      num_individuals: 2
    },
  ]
}
```

Example bad request response:

```
{
  status: 400
}
```

/link

In this operation, a user can manually create a group. They do this by manually specifying the case IDs that should compose the new group.

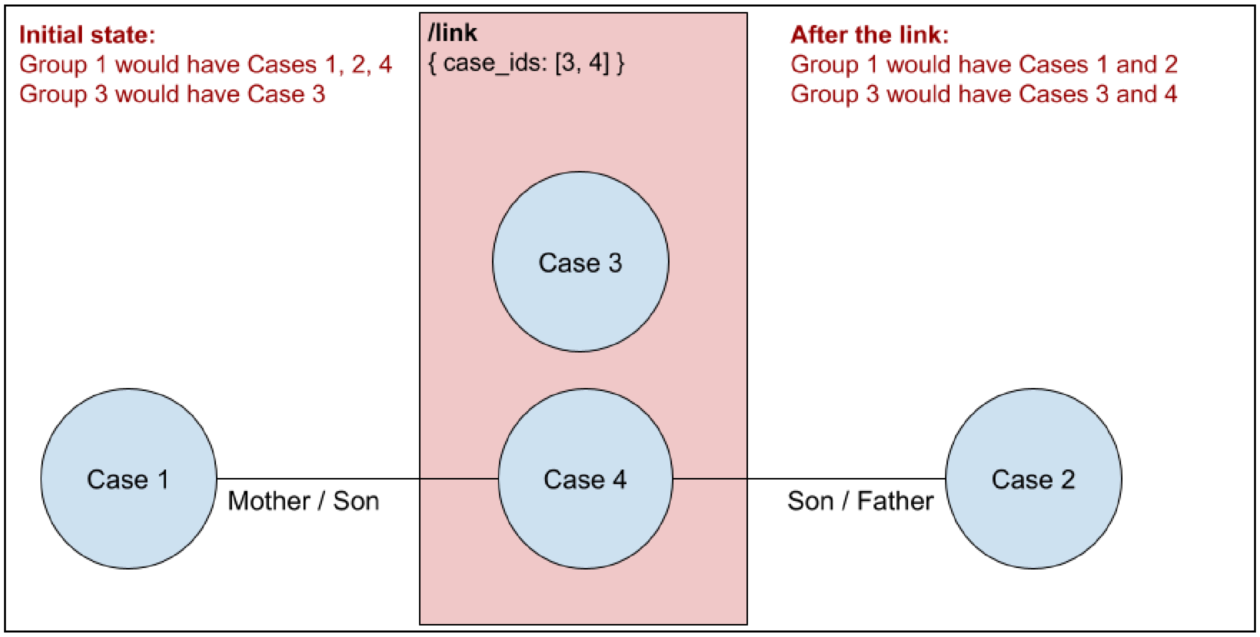
Each link request contains a list of case IDs that should comprise the new manually-created group. These case IDs are also considered removed from any previous group they were in.

If any non-existing case IDs are specified in the request, it should fail with `400` without modifying any state.

If successful, the response should include the newly created `group_id`.

Here are some more characteristics of a link operation:

- This manually created group (with the specified `case_ids`) should ignore any constraints imposed in problem 1. If `case_ids` is `1,2,3`, then the new group will contain cases 1, 2, and 3 assuming they exist.
- Similarly, any groups that were modified due to the `/link` request may (after the request) also violate the constraints imposed in problem 1. This is acceptable.
- This may mean that the group IDs change as modifications are processed.
- See the following visual example of a link:



Initial state:
Group 1 would have Cases 1, 2, 4
Group 3 would have Case 3
/link
{ case_ids: [3, 4] }
After the link:
Group 1 would have Cases 1 and 2
Group 3 would have Cases 3 and 4
Case 3
Case 1
Mother / Son
Case 4
Son / Father
Case 2

Example request body:

```
{
  case_ids: [3, 4]
}
```

Example success response:

```
{
  status: 200,
  new_group_id: 3
}
```

Example bad request response:

```
{
  status: 400
}
```

/unlink

Given a group ID, this operation splits each case in the group into their own individual groups (each with a single case). The result is equivalent to calling `/link` for each case within the given group ID.

If a non-existing group ID is specified in the request, it should fail with `400` without modifying any state.

If successful, the response should include the new group_ids that were created (in other words, the case_ids that formerly comprised the unlinked group).

Example request body:

```
{
  group_id: 3
}
```

Example success response:

```
{
  status: 200,
  new_group_ids: [3, 4, 5]
}
```

Example bad request response:

```
{
  status: 400
}
```

Problem 3: Adding Analytic Information

NOTE: You may be asked to modify your solution to problems 1 and 2 in a followup interview. We suggest you commit your solution to problems 1 and 2 before working on this problem. That way, you can easily rollback to avoid complicating the followup interview.

In order to improve the tool, we'd like to be able to track how the composition of case groups change over time (specifically, as the API operations in problem 2 are called).

Extend your solution from problem 2 to store this information. You just need to store the data, and you don't need to provide any means for accessing it. This problem is left intentionally vague. Please use the followup question to explain your solution.

The new data will be used purely for offline analytic purposes, and will never be used in the application's business logic.