# Algorithm Design

Jeremiah Navarro and Miles Claver

May 21, 2020

## 1 Algorithm Design

For our algorithm, the following bullet points may be considered.

- Make an integer programming (IP) algorithm that associates costs with each person (higher cost for teams with avoids in teammates, lower cost for prefers). In this case, we want to *minimize* cost. (Jeremiah will do this first)
- Make a simulated annealing (SA) algorithm that tries to find an *absolute maximum* "utility" by properly sorting teams. (No clue how this is done, yet). (Miles will do this first)

## 2 Timeline

A short timeline for the algorithm development is as follows.

1. We want to understand the algorithms (IP and SA) by May 22nd, ideally. This can go into Monday, but we want to have a solid grasp on our respective algorithms by Friday the 22nd.
2. We want to have a working implementation that can display the algorithms running properly, although not optimized, by May 29th. Two crucial features to get testing working are:
   (a) A JSON loader, which will load users preferences etc. from a JSON file.
   (b) A database loader, which calls the PostgreSQL database for user preferences.
3. Finally, we want to have some kind of optimizations and testing for the algorithm by the 5th of June. This way we can verify that the algorithm makes *sure* that nobody gets in teams with any avoids.

## 3 Introduction

### 3.1 Variables

We need to track the following variables for our algorithm. Anything that might lower the "cost" of our cost function/objective function is in red, while anything that would lower it is in green. Team sizes may be ambivalent depending on end implementation.

- User avoids,
- User prefers,
- Project preferences,
- Team sizes.

# 4 Algorithm, Version 0.1, Pass Two

Before we create the most general case of our needed algorithm, it may help to create a simpler case of the problem. Here, we only consider teams of fixed size (in the examples below, sizes of four), and we only consider preferences between individuals, without regard for project preferences.

We may represent project teams like so:

$$\{\{u_1, u_2, u_3, u_4\}, \{u_5, u_6, u_7, u_8\}, \{u_9, u_{10}, u_{11}, u_{12}\}\}$$

Where each $u_i$ indicates a user with ID $i$. Once we integrate this with our database, the ID will be whatever ID our CRUD application provides to the user. Similarly, our team set may be more compactly represented as:

$$\{T_1, T_2, T_3\}$$

Where each $T_i$ represents a particular team (i.e., $T_1 = \{u_1, u_2, u_3, u_4\}$. Since we are using a genetic algorithm to "evolve" the set of teams, which involves switching out users between teams, we consider the initial set of teams to be the first generation of our process, marked as $G_0$.

Externally, we provide a list of user preferences as well as user avoids between each other. In JSON, we may represent these lists as so:

```
{
        "prefers" : [[1, 2], [5, 7], [3, 4], [12, 1], [2, 1]],
        "avoids" : [[2, 1], [6, 11], [10, 3]]
}
```

These "edges" represent who prefers (or avoids) who. In the JSON above, `[5, 7]` in the "prefers" list indicates that student $u_5$ prefers student $u_7$. Similarly, `[2, 1]` in the "avoids" list indicates that student $u_2$ would like to avoid student $u_1$. Note that the prefers and avoids lists may be unordered (i.e. a set), but the sublists may not be, as they indicate the direction of preference/avoidance.

That said, the way we rank the fitness of a generation of project teams is by how many teams have students that prefer each other within their respective teams. For instance, if $T_1$ were to have a bidirectional 4-clique of students who prefer each other, then that team would contribute strongly to the fitness of that generation. Similarly, if $T_1$ were to have a bidirectional 4-clique of students who want to *avoid* each other, then $T_1$ would contribute strongly *against* the fitness of that generation.

We may create a very simple fitness function for these attributes below.