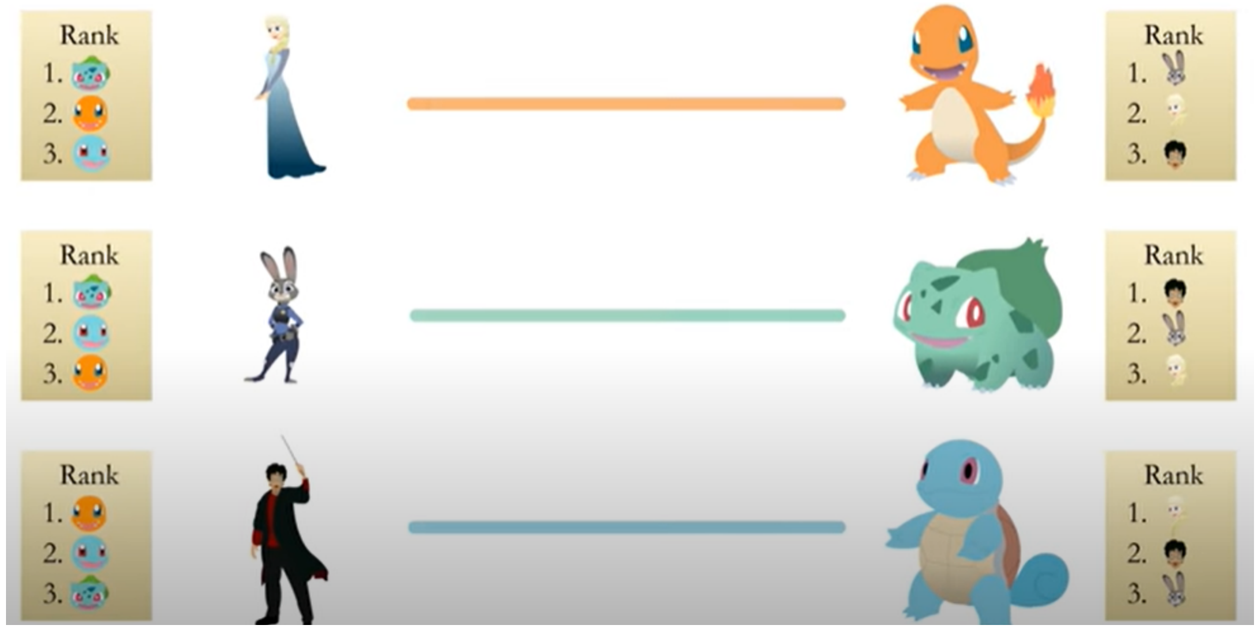
Introduction

Our project solves the stable matching problem. Given two sets, let's say of trainers and pokemon, each trainer will be paired with exactly one pokemon. Each trainer has an ordered list of preferences for which pokemon they want to be paired with, and each pokemon has a parallel list for trainers. In a stable matching there is no pair for which both a trainer and a pokemon would rather be with each other than with their assigned match. This creates a self-enforced pairing, since there is not a strong motivation for any trainer and pokemon to abandon their assigned match for someone else. Applications include matching dance partners, students with schools, or applicants with job positions.

In our implementation, the input sets of trainers and pokemon are represented with a bipartite graph. The first half of the nodes are trainers and the second half are pokemon. The edges map a trainer to a pokemon or a pokemon to a trainer, and the weight of the edge indicates ranking. So an edge from trainer 2 to pokemon 4 with a weight of 3 would indicate that trainer 2's third choice is pokemon 4.

Below is a visual example of a stable matching, continuing to use our example of trainers and pokemon. The left side is trainers and the right side is pokemon and the lines indicate stable matches. Screenshot from the video [Residency Match (Stable Marriage Problem, Gale-Shapley Algorithm)].



Implementation

We have an EdgeListGraph object to read the graph as a list of edges from an input file. This is then converted to a MatchingGraph object which runs the Gale-Shapley algorithm to determine which edges create a stable matching. These edges are then output to a file in the format of a Mathematica string, for visualization purposes.

To make implementing the Gale-Shapley algorithm easier, we represent our bipartite graph as a “Matching Graph.” This graph contains two lists: one of trainers and one of pokemon in the graph. The trainer list contains Trainer objects and the pokemon list contains Pokemon objects. We created these classes so that each trainer could easily keep track of their rankings of pokemon and vice versa. It is similar to an adjacency matrix since rankings are maintained in arrays, but with no wasted space (meaning it only keeps track of edges that are present in the graph). Hence it is also comparable to an adjacency list. The Trainer and Pokemon classes are separate since the two entities need distinct functions and have different roles to play in the algorithm. For example, Trainers must propose and Pokemon must accept or reject these proposals.

We use the Gale-Shapley algorithm to compute stable matchings. We learned this algorithm from the video referenced previously and from the Kleinberg & Tardos, Ch 1.1. A first problem: Stable Matching reading posted to Moodle. For the algorithm, we first add all of the trainers to a queue. In a while loop we go through each trainer in the queue. We look at a trainer’s highest ranked pokemon that the trainer has not yet proposed to. If that pokemon is free, we pair them. If the pokemon is not free and prefers to be with their pre-existing match, we add the current trainer to the end of the queue and move on to the next trainer. If the pokemon prefers the new trainer, we re-pair them with that trainer, add the pokemon’s pre-existing match back to the queue, and move on to the next trainer.

The Trainer and Pokemon rankings have reversed index roles to make our algorithm more efficient. For the trainers’ rankings, the index represents the ranking while the number at that index represents the pokemon. So ranking[0] = 5 means that this trainer’s first (0+1) choice is Pokemon 5. This is useful as a trainer successively proposes to pokemon in order of preference for them, since we can just go through the array sequentially. For the pokemons’ rankings, we instead have the index represent the trainer while the number at that index represents the ranking. So ranking[0] = 3 means that the pokemon ranks Trainer 0 as its third choice. This makes it easier to compare a pokemon’s rankings for two trainers to determine with whom they should match when proposed to but already engaged.

We tested our code thoroughly to ensure that it works. We created seven test input graphs and confirmed that the code properly produced a stable matching for each one.

User Manual

To use our solution, create an input graph in a .txt file as follows. The first line is the total number of vertices in the graph. If there are n vertices in the graph, the first n/2 represent the trainers and the last n/2 represent the pokemon. Keep this in mind as you add edges to your graph. After the line with the total number of vertices, each line should contain an edge. This will be represented by three numbers. For example, “1 4 2” would represent an edge from vertex 1 to vertex 4 with a weight of 2. Note that vertex indices start at 1 and end at n. For the code to run properly, the graph must have each trainer vertex rank each pokemon vertex (represented as a weighted edge from the trainer to the pokemon) and vice versa. Make sure there are no ties in ranking and that rank weights start at 1 and end with n/2 for each vertex ranking. Also, please include two blank lines at the end of the file after all the edges have been listed. Sample .txt graph files are shown in Data/GrfWTxt. In the same folder make a .grf file with the same name as your .txt file. This can be used to visualize the output of the algorithm in Mathematica, and must be included whether or not you use it so as to prevent the code from breaking. To create a Mathematica visualization add the same edges to the .grf file as you have in the .txt file. This .grf file should contain a Mathematica-formatted version of the graph. In addition to the edges, this will require coordinates for each vertex. These appear in lieu of the total number of vertices (since the total number of vertices is implied by the number of points you provide). Example .grf graph files are available in Data/GrfWTxt.

Once you have these files, you can run from the command line. Refer to the run.txt and stablematchingsdemos.txt in Java/UnitTests for more details. After compiling the code, running the main program requires two command line arguments. The first is the file path from which to read the input, which should be ../../Data/GrfWTxt/graph.txt, where graph.txt is the input graph you made for running the program. The second is the folder path to which to write the output (the edges which make the stable matching in the format of a Mathematica string). If you plan to use the Mathematica visualization, make your output destination ../../Data/GrfStb/.

Once you have run, if you wish to visualize using Mathematica, navigate out of the Java folder and to the Mathematica folder. Open the Mathematica file there and select Evaluation → Evaluate Notebook. You can change the input graph by using the input menu on the visualization. You can also toggle on and off various aspects of the visualization. Trainers are shown in purple, pokemon in yellow, and the edges which represent a stable matching in blue.

The classes and the functions are documented in the code.