Solving Variations of Stable Matching with Constraint Programming

Authors: Kevin He, Zora Mardjoko

Project Inspiration:

Our motivation comes from the frat/sorority Big-Little pairings on campus. One of Zora's frat friends just said how cool it would be if they had a Big-Little match that they could use for their frat, so that's basically why we decided to look into this topic - we thought it would be cool to basically build an app that others could use for their frat matchings, as well as research into exploring and building solutions to some additional variants.

Background Information:

In our research project, we started first looking at the Stable Matching Problem. This original problem exists in P, with the Gale-Shapley algorithm being a polynomial-time algorithm that can solve the stable-matching, essentially just outputs a matching in which there are no instabilities.

- Instabilities are defined as a pair of tuples (a,b) that have been matched together, where a is from A, and b is from B, where either a prefers some other b' in B which also prefers a in A compared to a', or b prefers some other a'' in A which also prefers b over b'' which it is currently paired with.

We begin by solving the Stable Matching problem through constraint programming, before progressing to the following variants:

1. **Stable Matching with Ties (SMT)**
2. **Stable Matching with Ties and Incompletes (SMTI)**
3. **Stable Matching with Incompletes As Optional Matchings (SMI2)**

Additionally, we explore another method that does not always guarantee stable matchings, but instead maximizes the sum of the preference scores. We call this method **Utilitarian Matching (UM).**

First, we decided to explore ties because often people may not be able to decide between two or more options; this is why we came up with our SMT approach.

Then, we decided to also explore how to deal with incomplete information, since very often people may know what their top 3 or top 5 options are, however, not care about the rest. To tackle this problem of incompletes, we decided to come up with two possible approaches that we thought were equally valid, the first being SMTI, and the second being SMI2, SMTI tackles the approach using what we have already built with ties, but assigning all those that were not included in a preference list the lowest tied preference rank, whereas SMI2 assumes that if a person was not included in another's matching, then the other person would rather be single than matched with this person.

Then, we thought that maybe we don't need to eliminate all instabilities, and that perhaps the overall happiness of the group of people should be optimized rather than just for nobody to be jealous or to cheat on their bigs or littles. After all, in a larger environment where there may be hundreds of bigs or littles, even having a couple of instabilities will probably not result in too many issues of cheating or other bigs hanging out with their non-littles. This was our motivation for Utilitarian Matching. Additionally, this method gave us the flexibility to weight the bigs' and littles' preferences in the score.

Codebase Overview

`biglittlematcher.py` - the big little matcher class that contains all the functions necessary to run SM, SMT, SMTI, SMI2, and UM
`residencymatcher.py` - the matcher class that contains the methods required to run SMC
`galeshapely.py` - the class that allows the user to run the Gale-Shapley algorithm
`app.py` - the code that runs the GS + Variations UI
`Solver.ipynb` - the Python notebook that tracks our development in code over the variations
`tester.ipynb` - the Python notebook that holds all our code for extensive testing

Implementation Details

The Stable Matching Problem

Inputs:
- `Bigs (Dict[str, Dict]), Littles (Dict[str, Dict]), Big_prefs (Dict[str, List]), Little_prefs (Dict[str, List])`

Description of Inputs:
- To be honest, the two variables Bigs and Littles are fairly irrelevant here, and while their type may be a bit confusing it is because we don't use them here but instead use them later on for other, more advanced variants. We don't need to use Bigs or Littles because Big_prefs and Little_prefs are already sufficient for us to complete everything we need.

Description of Algorithm:

- To properly set the constraints of the program, first, we need to declare the following 0/1 variable that tracks whether or not a big and a little are matched together, let us call this variable `x_{b}{l}`, then we can encode the following constraints:
    1. Each big must be matched to exactly one little
    2. Each little must be matched to exactly one big
    3. For every possible pair of big b or little l, at least one of the following must be true:
        a. Either b and l are matched together, OR b is matched with someone better than l, OR l is matched with someone better than b
- Note that it can never occur for all three conditions to be false, because this would imply an instability, since if b and l were both with l' and b', who they both preferred less than each other, then this is the definition of an instability.
- Note that since the Stable Matching problem is not an optimization problem but rather just a constraint satisfaction problem, we do not need to set any specific function to maximize, so in this case, we can simply tell the model to maximize the constant zero.

Stable Matching with Ties

Inputs:
- `Big_prefs (Dict[str, Dict[str, int]]), Little_prefs (Dict[str, Dict[str, int]])`

Description of Inputs:
- Now, we change the representation of big_prefs and little_prefs so that the values are now Dict[str, int], where the key is the name of the person's preferences, and the value is the rank of that preference.
- By using this representation, we can store equally ranked preferences with the same integer value, and can quickly get the rank of a person in a preference list by dictionary lookup.
- This approach seemed to faster than using `Dict[str, List[Optional[List, int]]` representation, since this List representation of someones preferences doesn't exactly tell us where a person is ranked in the list, and would have to require some O(n) time index function to get a person's rank given their name.

Description of Algorithm:
- Again, we pretty much encode the following constraints as regular Stable Matching, except with the following change in the code:

    3. For every possible pair of big b or little l, at least one of the following must be true:
        a. Either b and l are matched together, OR b is matched with someone either better than or equally preferred to l, OR l is matched with someone either better or equally preferred than b.
- Note that again, that all three cannot be false, since this would imply that b and l are again matched with someone worth more for them than each other, thus resulting in an instability.

- Again, no optimization necessary, since any solution found not violating the constraints is a valid solution to the Stable Matching with Ties problem, so we can just set the arbitrary maximizing function to zero.

Stable Matching with Ties and Incompletes

Inputs:
- `Big_prefs (Dict[str, Dict[str, int]]), Little_prefs (Dict[str, Dict[str, int]])`

Description of Inputs:
- Same as with the SMT solution, for the same reason that dictionary lookup is faster than list index lookup.

Description of Algorithm:
- For this variation of the algorithm, we assume that everyone who opts to be matched, even with an incomplete preference list, must end up being matched with someone. Additionally, we assume that the reason they did not include some people on their preference list is that these people are even lower ranked than the lowest person they ranked on their current incomplete preference list.
- Therefore, by these assumptions, simply by setting their rankings all to be the `max_rank`, and then running essentially our SMT algorithm as above should lead to a valid solution to this variation of the SMTI problem.

Stable Matching with Incompletes (Variant 2)

Inputs:
- `Big_prefs (Dict[str, Dict[str, int]]), Little_prefs (Dict[str, Dict[str, int]])`

Description of Inputs:
- Same as above!

Description of Algorithm:
- This variation differs from the previous one in that we assume people would rather be unmatched than matched with someone they did not rank. Thus, we modify the matching constraints to allow each participant to be matched at most once rather than exactly once.

Utilitarian Matching

Inputs:
- `Big_prefs (Dict[str, List[str]]), Little_prefs (Dict[str, List[str]]), preference_weight (float), enforce_exactly_one (bool)`

Description of Inputs:

- The preference list inputs to this algorithm are different from the previous algorithms, using lists instead of dicts.
- The `preference_weight` variable allows the user to specify whether they want the bigs' preferences or the littles' preferences to be weighted more in the calculation. Values closer to 0 will weight the littles' preferences more, and values closer to 1 will weight the bigs' preferences more. A value of 0.5 is equally weighted with both.
- The `enforce_exactly_one` variable can allow for multiple matches (aka multiple bigs to one little, or one big having multiple littles)
    - When this variable is false, there is a max field that allows bigs/littles to specify their personal maximum limit

Description of Algorithm:

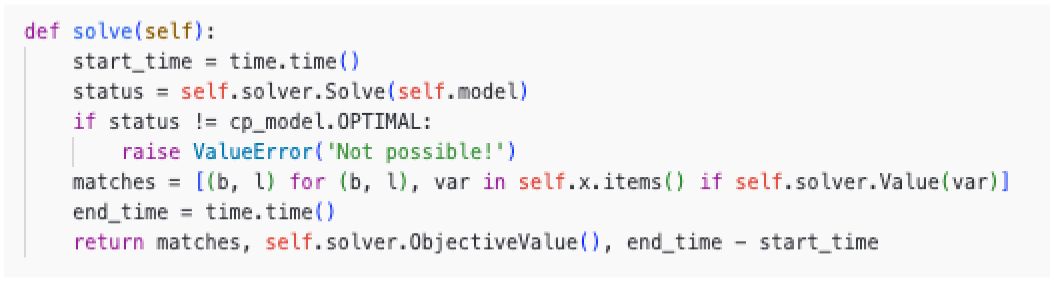
- The algorithm constructs an optimization model that selects matches to maximize the sum of these scores across all pairs.
- Unlike stable matching algorithms, utilitarian matching does not enforce stability or fairness constraints, so it may result in outcomes where some participants would prefer to deviate if given the chance.

Additional Class Helper Functions

Stability Check Helper

Unfortunately, due to poor planning and lack of foresight, we ended up having to use two different methods that check for instabilities due to our dual representation of preferences in some cases as a List, and in other cases a Dictionary. This ended up causing a lot of confusion, however, the fundamental underlying checks between both methods are the same. That is, for every possible pair (b,l), if b and l are not matched together, then check that they would rather be with whom they are currently matched with, than with each other.

Solver Helper



```python
def solve(self):
    start_time = time.time()
    status = self.solver.Solve(self.model)
    if status != cp_model.OPTIMAL:
        raise ValueError('Not possible!')
    matches = [(b, l) for (b, l), var in self.x.items() if self.solver.Value(var)]
    end_time = time.time()
    return matches, self.solver.ObjectiveValue(), end_time - start_time
```

This solver just solves the model with the constraints that we have already set with our build_model methods, and then outputs then list of matches, as well as the objective value that was achieved by the solver, and the amount of time it took for this process to complete (we use this duration variable later to help us compare runtimes of the different models).

Pretty Print Helper

```python
def pretty_print(self):
    print(self.solver.ResponseStats())
    score_achieved = self.solver.ObjectiveValue()
    print(f"Total preference score: {score_achieved:.0f}")
    COLORS = ['aqua', 'coral', 'darkgreen', 'gold', 'darkolivegreen1',
              'deeppink', 'crimson', 'darkorchid', 'bisque', 'yellow']
    G = graphviz.Graph()
    for (b, l), var in self.x.items():
        if self.solver.Value(var):
            # Use default penwidth of 1 if scores dict doesn't exist or doesn't have the pair
            penwidth = str(self.scores.get((b, l), 1)) if hasattr(self, 'scores') else "1"
            G.edge(f'{b}', f'{l}', penwidth=penwidth)
            G.node(f'{b}', color=COLORS[hash(b) % len(COLORS)])
            G.node(f'{l}', color=COLORS[hash(l) % len(COLORS)])
    display(G)
```

The pretty print helper essentially prints out all the stats from the ortools solver, as well as graphs the output matching between people. You may see the penwidth feature, this is to help the Utilitarian Matching display; it just determines the width of the edges between nodes of the graph, but by default, we set it to 1.

Experimental Plan, Hypotheses + Design

1. SM

For Regular Stable Matching, we wanted to experiment on regular Gale-Shapley vs a constraint programming version of the problem. To do this, we will generate random tests (random preference lists) and time how long our solver takes. Our hypothesis: we expect our constraints model to be slower than Gale-Shapley because GS is an efficient algorithm that focuses specifically on solving this problem whereas our constraints model is a more general model that can solve other problems.

2. SMT

We are similarly interested in the speed of SMT vs regular stable matching. Does the introduction of ties in the preference lists lead to a slow down or speed up in the algorithm?

3. SMTI

For Stable Matching with Ties and Incompletes, we wanted to compare the speeds between SMTI with SMT, and SM. We hypothesize that SMTI will probably have a faster time, because

4. SMI2

For Stable Matching with Incompletes 2nd Variation, we want to see if it is faster than SMTI, since there are less matches to be made.

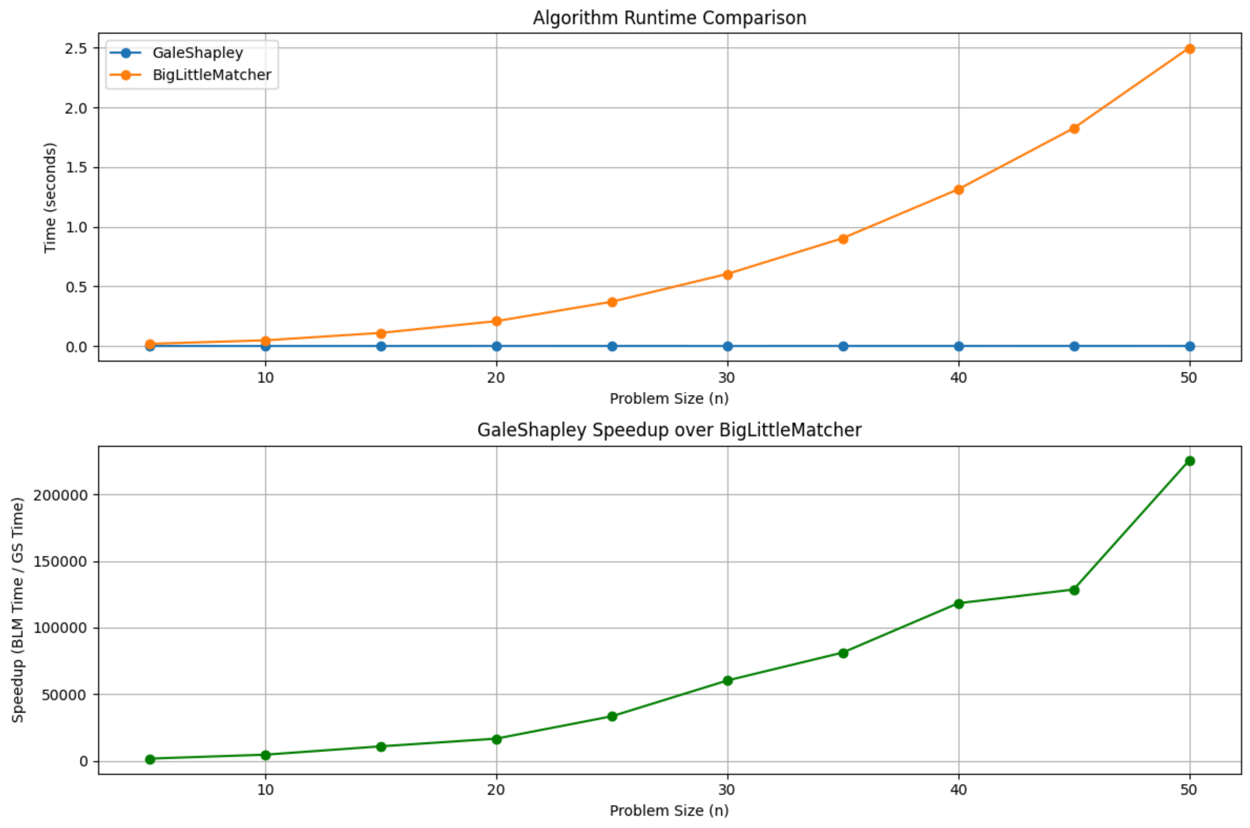
5. UM

For Utilitarian Matching, we want to compare results between UM and SM, see if there are any differences/similarities we can draw, and then move on to play around with the weights and twin matchings. One Hypothesis is that setting the weights to 0.5 (by equally valuing both sides), can lead to stable matchings. Another hypothesis is that non 0.5 weights will result in instabilities.

Experimental Results + Analysis

Stable Matching

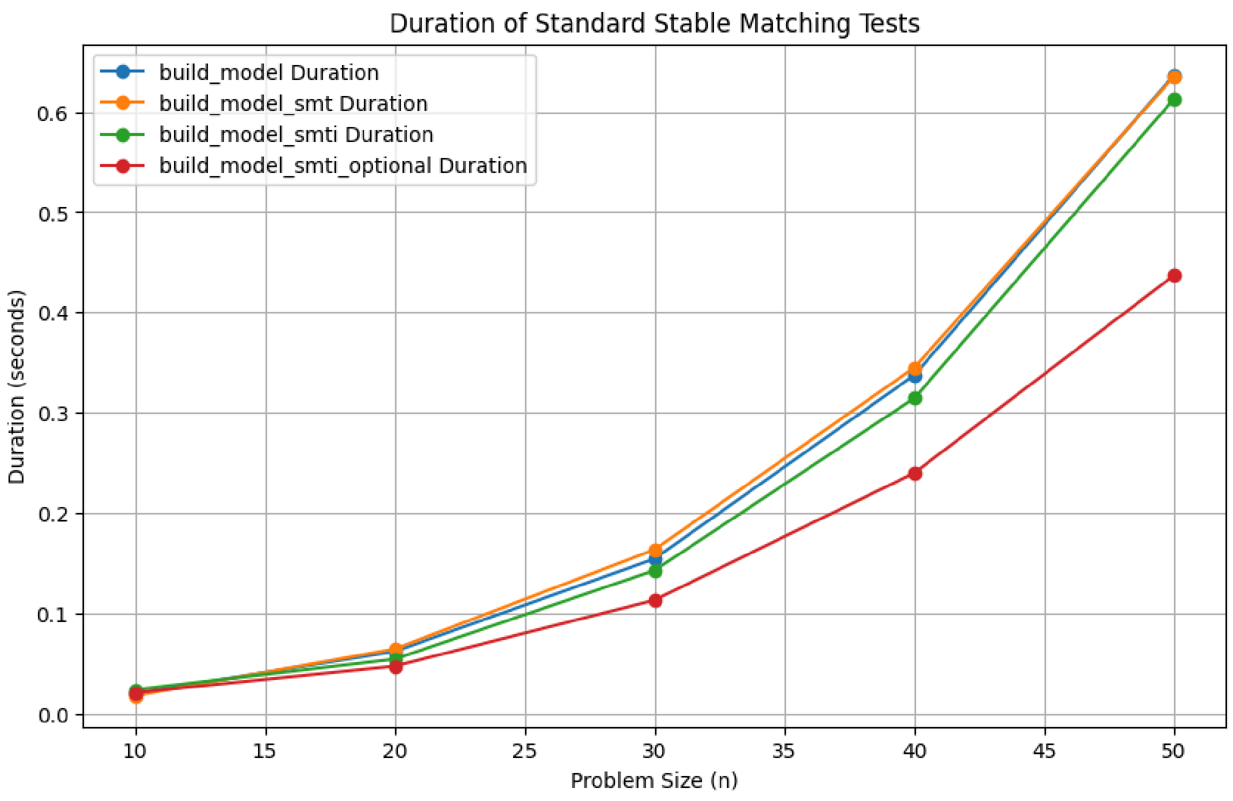
The first interesting thing we found is that GS is much faster than our CP solver on the stable matching problem.



This confirmed our initial hypothesis. Interestingly, Gale Shapley had an exponential speedup over our CP based solution as the input size grew.

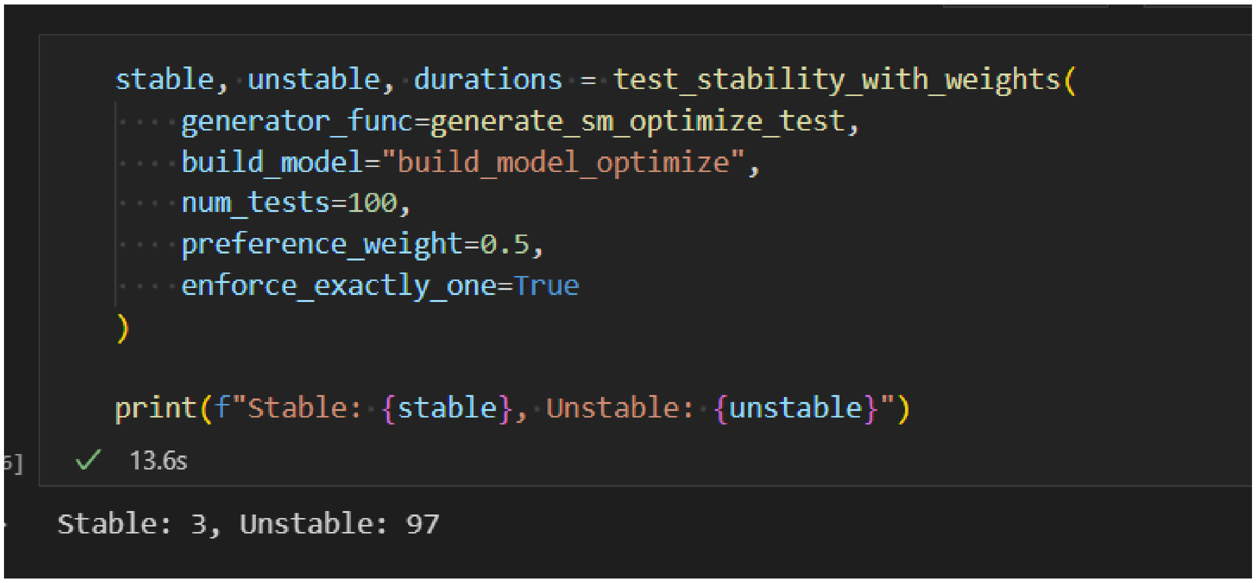
SMT, SMTI, SMI2

In our tests, we focused on comparing the duration of time to solve various sized inputs. The comparisons are seen below, in the graph. As we expected, when getting matched was optional, the time decreased.



UM

We wanted to figure out if UM results in stable matchings. From our test (results shown below), this is not the case.



```
stable, unstable, durations = test_stability_with_weights(
    generator_func=generate_sm_optimize_test,
    build_model="build_model_optimize",
    num_tests=100,
    preference_weight=0.5,
    enforce_exactly_one=True
)

print(f"Stable: {stable}, Unstable: {unstable}")
```

```
Stable: 3, Unstable: 97
```

This was an interesting result, as most of the tests resulted in pairings with instabilities. We did some deeper exploration, and a simple example shows why maximizing preference score sum does not always result in a stable matching.

Example

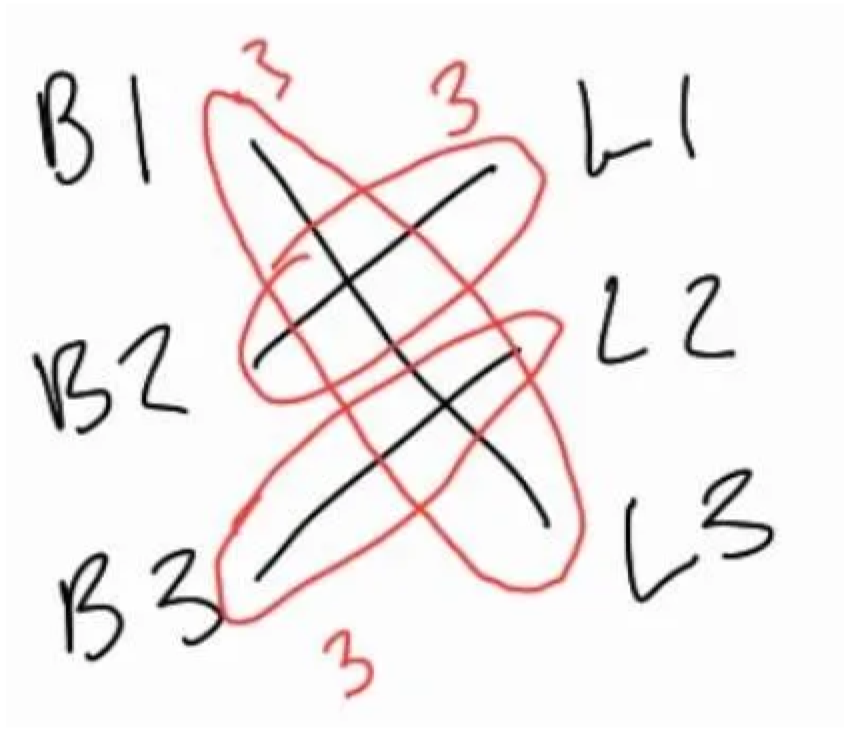
Bigs' Preferences:
  B1: ['L3', 'L1', 'L2']
  B2: ['L1', 'L3', 'L2']
  B3: ['L3', 'L2', 'L1']

Littles' Preferences:
  L1: ['B3', 'B2', 'B1']
  L2: ['B3', 'B1', 'B2']
  L3: ['B3', 'B1', 'B2']

Pairings:
  B1 ↔ L3
  B2 ↔ L1
  B3 ↔ L2

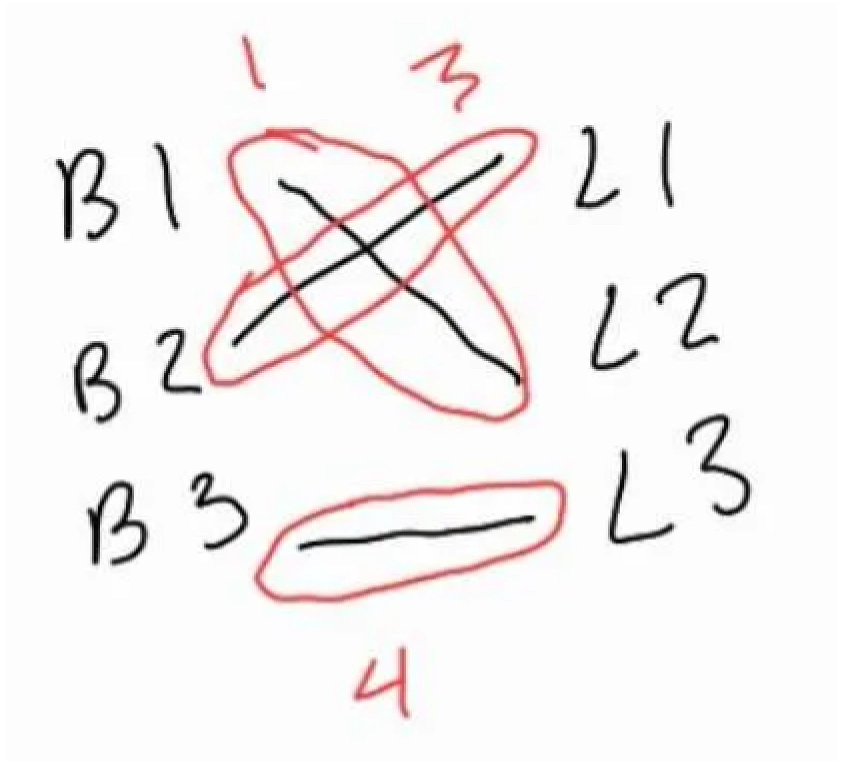
In the above pairings, the final pairings represent the output of UM. However, an instability is the pair B3 - L3. We can see that B3 prefers L3 over L2 (its current partner), and L3 prefers B3 over its current partner B1.

Let’s analyze the score for the UM output.



Using the formula (max rank - rank of big) + (max rank - rank of little) for each pair (which is what our solver uses), we get the above calculation. The total sum then becomes 9.

Now, let's analyze the score for the stable matching.



Using the same formula, the total sum is 8.

So, the pairings with stable matching results in a lower score than the pairings with an instability. Again, this makes sense because in larger groups, say big-little pairings for 40 people, maybe maximizing overall utility makes more sense than a stable matching.

Future Areas of Exploration

We originally considered the problem of Stable Matching with Couples, since it required a different approach than the other stable matching problems. However, we did not implement this for this project, but it is a potential area of future research.