Stable Matching Problems

Ahmed Luqman - 24100041

April 2024

1 Introduction

The Standard Gale-Shapely algorithm is used to match members of two different sets. Perhaps the most notable example of its application is the Stable marriage problem which deals with setting up males and females in such a way that no *instability*[1] is left in the final matching. The Gale-shapely algorithm is guaranteed to return a stable matching and does so in order $O(n^2)$ where n is the number of pairs you wish to form in the final matching. There is a somewhat related problem of matching roommates with one another. The Gale Shapely algorithm does not guarantee a stable matching for this problem. The Stable marriage and Stable roommate problem differ in that the first problem pairs members of two different sets where as the second problem aims to pair members of the same set. The stable roommates problem also has an efficient algorithm which was introduced by David Irving in his paper "An Efficient Algorithm for the 'Stable Roommates' Problem" in 1985, it finds a stable matching (if a matching exists)[2]. In this paper we extend this idea and look at Matchings that involve multiple sets, we do not try to come up with algorithms that guarantee solutions, rather to try to understand the problem and come up with matchings that are in some sense *acceptable*.

2 Proof of Correctness and Termination of Gale-Shapley Algorithm

Correctness

1. **Definition of Stable Matching**: A matching is stable if there are no two individuals who both prefer each other over their current partners. Formally, there are no "blocking pairs."

2. **Initial Matching**: Initially, all vertices are free. Each proposer proposes to their top-choice vertex, and each vertex responds "maybe" to their favorite proposer. This initial matching might not be stable, as there could still be blocking pairs.

3. **Proposals and Rejections**: As the algorithm progresses, proposers continue to propose to vertices on their preference list, and vertices either accept or reject proposals based on their preferences. If a vertex receives a proposal from a proposer they prefer over their current partner, they accept that proposal and reject all other proposals.

4. **Termination Condition**: The algorithm continues until there are no more rejections. At this point, all vertices have been matched in a stable manner, with no blocking pairs remaining.

5. **Stability**: At termination, there can be no blocking pairs, as any pair of individuals would have either rejected each other or be matched to each other, satisfying the stability condition.

Termination

1. **Proposal Process**: In each round of the algorithm, proposers make proposals to vertices according to their preference lists. Since the number of proposers and vertices is finite, there can only be a finite number of proposals made.

2. **Rejection Process**: Similarly, vertices can only reject a finite number of proposals, as they can only receive a finite number of proposals.

[1]A male and female pair who would both prefer to have been married to each other instead of their current partners.
[2]Note that for the Stable roommates problem a stable matching is not guaranteed to exist for every possible set of preference lists.

3. **Termination Condition**: The algorithm terminates when there are no more rejections, indicating that all vertices are matched. Since the number of rejections is finite, the algorithm must terminate after a finite number of steps.

Thus, the Gale-Shapley algorithm is both correct, as it produces stable matchings, and terminates in a finite number of steps.

3 Time Analysis for Stable Matching Generation of Various Sizes

We will generate all possible combinations of matchings possible of various sizes and check the time taken to compute all stable matchings. The code follows the following structure:

Algorithm 1 Gale Shapely Time Computation for all Preference List Combinations of size n

```
function ComputeAllMatchings(n)
    P1=Generate all permutations of size n for set 1
    P2=Generate all permutations of size n for set 2
    Combinations1=Generate all combinations between all sets in P1
    Combinations2=Generate all combinations between all sets in P2
    Start time
    for c1 in Combinations1 do
        for c2 in Combinations2 do Compute Stable matching of c1 and c2 using Gale Shapely
        end for
    end for
    End Time and return Time taken
end function
```

We generated all possible preference lists of size 3,4 and 5 using the algorithm above. Our code ran in under 0.5 second for n = 3, for 4-5 seconds for n = 4 but crashed for n = 5 because of the size of the input and data. n = 5 gives nearly $5^{10} \approx 10$ million lists which caused the system to run out of space (See Figure 1).

4 The Stable Roommate Problem and 1 Dimensional Matching

The Stable Roommates Problem is a classic problem in the field of matching theory, aiming to pair individuals into stable pairs while respecting their preferences. It involves a set of individuals, each ranking a subset of others as potential partners. The goal is to pair them such that no two individuals prefer each other over their current partners

The algorithm consists of two phases. In Phase 1, participants propose to each other, in a manner similar to that of the Gale-Shapley algorithm for the stable marriage problem. Each participant orders the other members by preference, resulting in a preference list—an ordered set of the other participants. Participants then propose to each person on their list, in order, continuing to the next person if and when their current proposal is rejected. A participant will reject a proposal if they already hold a proposal from someone they prefer. A participant will also reject a previously-accepted proposal if they later receive a proposal that they prefer. In this case, the rejected participant will then propose to the next person on their list, continuing until a proposal is again accepted. If any participant is eventually rejected by all other participants, this indicates that no stable matching is possible. Otherwise, Phase 1 will end with each person holding a proposal from one of the others.

Consider two participants, q and p. If q holds a proposal from p, then we remove from q's list all participants x after p, and symmetrically, for each removed participant x, we remove q from x's list, so that q is first in p's list; and p, last in q's, since q and any x cannot be partners in any stable matching. The resulting reduced set of preference lists together is called the Phase 1 table. In this table, if any reduced list is empty, then there is no stable matching. Otherwise, the Phase 1 table is a stable table. A stable table, by definition, is the set of preference lists from the original table after members have been removed from one or more of the lists, and the following three conditions are satisfied (where reduced list means a list in the stable table),

- p is first on q's reduced list if and only if q is last on p's
- p is not on q's reduced list if and only if q is not on p's if and only if q prefers the last person on their list to p; or p, the last person on their list to q

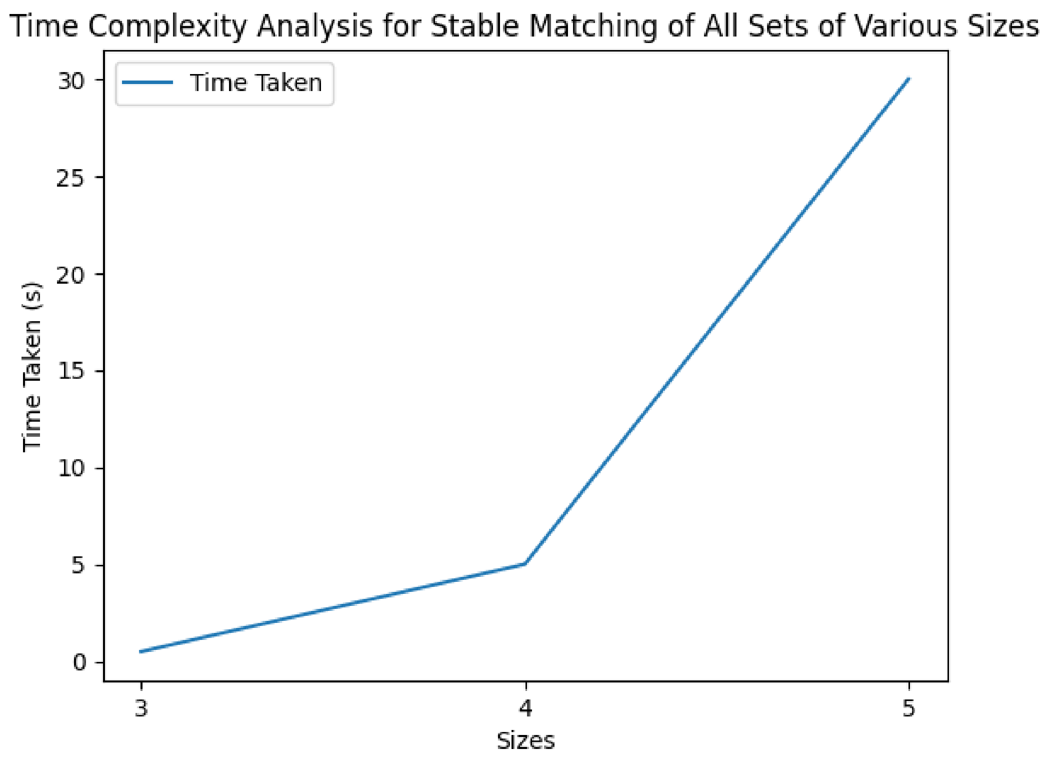


Figure 1: Time Analysis of sets of various Sizes, Do note, the time taken for size=5 is much greater than 30s

- no reduced list is empty

```
T = Phase 1 table;
while (true) {
    identify a rotation r in T;
    eliminate r from T;
    if some list in T becomes empty,
        return null; (no stable matching can exist)
    else if (each reduced list in T has size 1)
        return the matching M = {{x, y} | x and y are on each other's lists in T}; (this is a stable
matching)
}
```

Figure 2: Phase 2 of the algorithm can now be summarized as follows

4.1 Irving's Algorithm

Irving's Algorithm is an efficient solution to the Stable Roommates Problem. It operates by allowing individuals to propose to their preferred partners iteratively until stable pairings are achieved. The algorithm guarantees convergence to a stable matching and has a worst-case time complexity of $O(n^2)$, making it efficient for moderate-sized instances.

In comparison to the Stable Marriage Problem:

- **Stable Roommates Problem**: Irving's Algorithm is commonly used to solve the Stable Roommates Problem. This algorithm allows individuals to iteratively propose to their preferred roommates until stable pairings are achieved.

- **Stable Marriage Problem**: The Gale-Shapley Algorithm, also known as the Deferred Acceptance Algorithm, is typically used to solve the Stable Marriage Problem. This algorithm allows individuals from one set to propose to individuals from the other set based on their preferences, and each individual accepts or rejects proposals iteratively until stable marriages are formed.

- Both problems involve forming stable pairs based on individuals' preferences, but the Stable Roommates Problem deals with forming pairs within a single set, while the Stable Marriage Problem involves forming pairs between two disjoint sets.

- In the Stable Roommates Problem, each individual ranks all others within the same set, while in the Stable Marriage Problem, each individual ranks members of the opposite set.

- The Stable Marriage Problem has a more rigid structure with two distinct sets and the requirement for balanced pairings, while the Stable Roommates Problem allows for more flexibility as it involves forming pairs within a single set.

Unlike the stable marriage problem, a stable matching may fail to exist for certain sets of participants and their preferences. For a minimal example of a stable pairing not existing, consider 4 people A, B, C, and D, whose rankings are:

$$A : (B, C, D), B : (C, A, D), C : (A, B, D), D : (A, B, C)$$

In this ranking, each of A, B, and C is the most preferable person for someone. In any solution, one of A, B, or C must be paired with D and the other two with each other (for example AD and BC), yet for anyone who is partnered with D, another member will have rated them highest, and D's partner will in turn prefer this other member over D. In this example, AC is a more favorable pairing than AD, but the necessary remaining pairing of BD then raises the same issue, illustrating the absence of a stable matching for these participants and their preferences.

5 One step up: 3 dimensional matchings

5.1 Extending the Gale Shapely algorithm

Let us define the following problem:

As an instructor, you need to sort students into groups of 3 for a course project, however you require that the groups be formed in such a way that there is exactly one graduate student, one Senior year student and one Junior year student in each group of three.

So instead of one preference list, you get back two preference lists for each individual, take an example of a Junior year student, his preference list may look something like this:

Type	1st	2nd	3rd
Senior-Year	2	3	1
Graduate	1	2	3

Table 1: Example of what a Junior year student's list may look like

One issue that we need to take care of before even thinking of how to find the optimal group allocation is to check if an optimal group allocation even exists in the first place. The guarantee for the Stable Marriage Problem is due to the properties of bipartite graphs, this guarantee does not hold for the 1-dimensional case.

We know that the Gale-Shapely Algorithm guarantees a stable matching for pair-wise matching from two different sets. What if we break down our 3 dimensional matching into a 2 + 1 dimensional matching. Assume that when we get a preference list as shown in Table 1, we only use the first list and create pairs of students from Junior year and Senior year, once we have these stable pairs, we somehow merge the preference lists for both the juniors and seniors for the graduates.

We can do this my simply adding up the positions of a graduate student on both lists. Assume any pair of Junior and Senior students give us the following lists:

Student	1st	2nd	3rd
Junior-Year	2	1	3
Senior-Year	1	2	3

Table 2: Two Lists for the Graduate students rankings

	1st	1st	2nd
Merged List	1	2	3

Table 3: Two Lists for the Graduate students rankings

If we merge them we get the following:
The students are indifferent towards Graduate student 1 and 2 because one of them is ranked 1st in ones preference list and 2nd in the others, so the overall satisfaction of the group is the same if either of these two join as the third member.

Now we need to modify the Graduate students preference list as well. It will be of the form:

Type	1st	2nd	3rd
Junior-Year	3	1	2
Senior-Year	2	1	3

Table 4: Two Lists for the Graduate student's rankings of each other member

We can also merge these by looking at all existing groups and ranking them using the two individual preference lists, so if there are three groups (1,3), (2,1), and (3,2), using the lists given in the above we can create the following adjusted list:

Type	1st	2nd	2nd
Group	(3,2)	(1,3)	(2,1)

and now we can once again use the standard gale shapely algorithm by using these merged preference lists. A problem arises when the third group member is added though. Even though the initial two person groups were stable, there is a possibility that either of the two existing group members along with the new graduate student now form an instability.

Take the following generalized example,

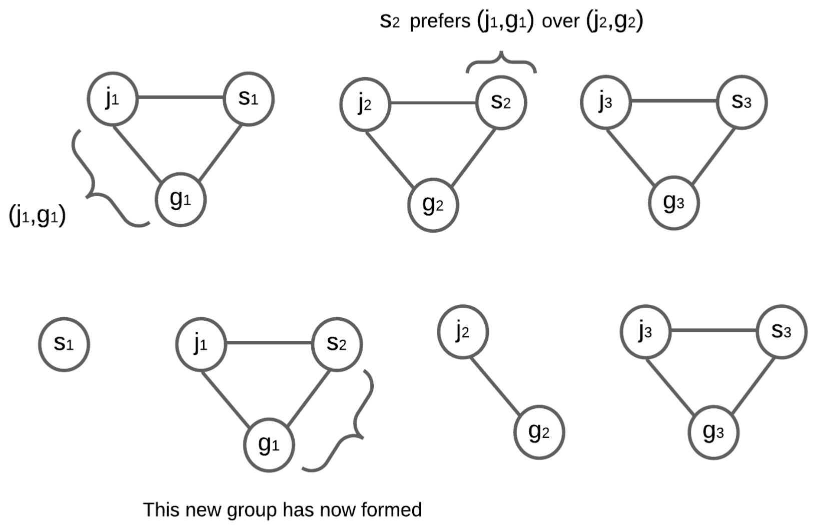
Let j_1, s_1 and g_1 denote a junior, senior and, graduate student respectively from group 1. Upon our initial run of the Gale-Shapely algorithm to form the pairs, j_1 and s_1 were paired up, along with j_2 and s_2 and j_3 and s_3. Now after the addition of Graduate students, s_2 prefers j_1 and g_1 as a pair, even though for s_2, $j_2 > j_1$, but now as a whole, given a merged preference list, $(j_1, g_1) > (j_2, g_2)$. If (j_1, g_1) also prefer $s_2 > s_1$ this will form an instability and the final matching is therefore not stable. So the Gale-Shapely algorithm cannot be extended or modified to cater to three dimensional matching.

Algorithm 2 Gale-Shapley Algorithm attempt for 3-Dimensional Matching

```
 1: Initialize all students and their preference lists
 2: Initialize empty lists for matched groups and unmatched students
 3: while there are unmatched Senior year students do
 4:     Select an unmatched Senior-Year student $J$
 5:     Let $S$ be the highest-ranked Senior-Year student in $J$'s preference list who hasn't rejected $J$ yet
 6:     if $S$ is unmatched then
 7:         Add group $(J, S)$ to the matched groups list
 8:     else
 9:         if $S$ prefers $J$ to their current Junior partner then
10:             Leave the old partner and join the group with $J$
11:         end if
12:     end if
13: end while
14: Merge preference lists of both $(J, S)$ and $G$
15: while there are unmatched Graduate students do
16:     Select an unmatched Graduate student $G$
17:     Let $G$ be the highest-ranked Graduate student in $(J, S)$'s preference list who hasn't rejected $(J, S)$ yet
18:     if $G$ is unmatched then
19:         Add group $(J, S, G)$ to the matched groups list
20:     else
21:         if $G$ prefers $(J, S)$ to their current group then
22:             Leave the old group and join the group with $(J, S)$
23:         end if
24:     end if
25: end while
```

Take the following generalized example,

Let j_1, s_1 and g_1 denote a junior, senior and, graduate student respectively from group 1. Upon our initial run of the Gale-Shapely algorithm to form the pairs, j_1 and s_1 were paired up, along with j_2 and s_2 and j_3 and s_3. Now after the addition of Graduate students, s_2 prefers j_1 and g_1 as a pair, even though for s_2, $j_2 > j_1$, but now as a whole, given a merged preference list, $(j_1, g_1) > (j_2, g_2)$. If (j_1, g_1) also prefer $s_2 > s_1$ this will form an instability and the final matching is therefore not stable. So the Gale-Shapely algorithm cannot be extended or modified to cater to three dimensional matching.



5.2 Existence of a stable matching

To show that it is not guaranteed that a stable matching exists for every given initial configuration (preference lists), we can try to prove that a best response cycle exists in a given 3-D matching.
Let us look at an example setup:

j_1 :

Type	1st	2nd	2nd
senior	1	3	2

Type	1st	2nd	2nd
graduate	2	1	3

j_2 :

Type	1st	2nd	2nd
senior	1	2	3

Type	1st	2nd	2nd
graduate	3	1	2

j_3 :

Type	1st	2nd	2nd
senior	2	3	1

Type	1st	2nd	2nd
graduate	1	2	3

s_1 :

Type	1st	2nd	2nd
junior	1	2	3

Type	1st	2nd	2nd
graduate	2	1	3

s_2 :

Type	1st	2nd	2nd
junior	3	1	2

Type	1st	2nd	2nd
graduate	3	2	1

s_3 :

Type	1st	2nd	2nd
junior	1	3	2

Type	1st	2nd	2nd
graduate	2	1	3

g_1 :

Type	1st	2nd	2nd
junior	1	3	2

Type	1st	2nd	2nd
senior	1	2	3

g_2 :

Type	1st	2nd	2nd
junior	2	1	3

Type	1st	2nd	2nd
senior	3	2	1

g_3 :

Type	1st	2nd	2nd
junior	1	2	3

Type	1st	2nd	2nd
senior	1	2	3

Let us look at the nine possible groupings that can be formed:

1. $(j_3, s_2, g_1), (j_1, s_3, g_2), (j_2, s_1, g_3)$
2. $(j_3, s_2, g_1), (j_1, s_3, g_3), (j_2, s_1, g_2)$
3. $(j_3, s_2, g_1), (j_1, s_1, g_3), (j_2, s_3, g_2)$
4. $(j_3, s_2, g_2), (j_1, s_3, g_1), (j_2, s_1, g_3)$
5. $(j_3, s_2, g_2), (j_1, s_3, g_3), (j_2, s_1, g_1)$
6. $(j_3, s_2, g_2), (j_1, s_1, g_3), (j_2, s_3, g_1)$
7. $(j_3, s_2, g_3), (j_1, s_3, g_1), (j_2, s_1, g_2)$
8. $(j_3, s_2, g_3), (j_1, s_3, g_2), (j_2, s_1, g_1)$
9. $(j_3, s_2, g_3), (j_1, s_1, g_2), (j_2, s_3, g_1)$

To show that all of these matchings are unstable we can first start my splitting them into Individual and Groupwise pairings:

	1st	2nd	3rd	4th	5th	6th	7th	8th	9th
j1	$(s3, g2)$	$(s3, g3)$	$(s2, g2)$	$(s3, g1)$	$(s2, g3)$	$(s1, g2)$	$(s2, g1)$	$(s1, g3)$	$(s1, g1)$
j2	$(s2, g2)$	$(s2, g1)$	$(s1, g2)$	$(s2, g3)$	$(s1, g1)$	$(s3, g2)$	$(s1, g3)$	$(s3, g1)$	$(s3, g3)$
j3	$(s1, g1)$	$(s1, g2)$	$(s3, g1)$	$(s1, g3)$	$(s3, g2)$	$(s2, g1)$	$(s3, g3)$	$(s2, g2)$	$(s2, g3)$
s1	$(j1, g3)$	$(j1, g2)$	$(j2, g3)$	$(j1, g1)$	$(j2, g2)$	$(j3, g3)$	$(j2, g1)$	$(j3, g2)$	$(j3, g1)$
s2	$(j3, g3)$	$(j3, g2)$	$(j1, g3)$	$(j3, g1)$	$(j1, g2)$	$(j2, g3)$	$(j1, g1)$	$(j2, g2)$	$(j2, g1)$
s3	$(j2, g1)$	$(j2, g3)$	$(j3, g1)$	$(j2, g2)$	$(j3, g3)$	$(j1, g1)$	$(j3, g2)$	$(j1, g3)$	$(j1, g2)$
g1	$(j3, s1)$	$(j3, s3)$	$(j2, s1)$	$(j3, s2)$	$(j2, s3)$	$(j1, s1)$	$(j2, s2)$	$(j1, s3)$	$(j1, s2)$
g2	$(j2, s3)$	$(j2, s1)$	$(j3, s3)$	$(j2, s2)$	$(j3, s1)$	$(j1, s3)$	$(j3, s2)$	$(j1, s1)$	$(j1, s2)$
g3	$(j3, s2)$	$(j3, s3)$	$(j1, s2)$	$(j3, s1)$	$(j1, s3)$	$(j2, s2)$	$(j1, s1)$	$(j2, s3)$	$(j2, s1)$

And Groupwise preferences:

Pair	**Preferences**	**Pair**	**Preferences**
(j1, s1)	[2, 3, 1]	(j1, s2)	[2, 3, 1]
(j1, s3)	[1, 2, 3]	(j1, g1)	[3, 1, 2]
(j1, g2)	[3, 1, 2]	(j1, g3)	[2, 3, 1]
(j2, s1)	[2, 3, 1]	(j2, s2)	[2, 3, 1]
(j2, s3)	[1, 2, 3]	(j2, g1)	[1, 2, 3]
(j2, g2)	[1, 2, 3]	(j2, g3)	[2, 1, 3]
(j3, s1)	[1, 2, 3]	(j3, s2)	[1, 2, 3]
(j3, s3)	[1, 2, 3]	(j3, g1)	[1, 3, 2]
(j3, g2)	[1, 3, 2]	(j3, g3)	[1, 2, 3]
(s1, j1)	[1, 2, 3]	(s1, j2)	[1, 2, 3]
(s1, j3)	[1, 2, 3]	(s1, g1)	[1, 2, 3]
(s1, g2)	[2, 1, 3]	(s1, g3)	[1, 3, 2]
(s2, j1)	[3, 1, 2]	(s2, j2)	[1, 2, 3]
(s2, j3)	[1, 3, 2]	(s2, g1)	[3, 1, 2]
(s2, g2)	[3, 2, 1]	(s2, g3)	[3, 1, 2]
(s3, j1)	[2, 3, 1]	(s3, j2)	[2, 1, 3]
(s3, j3)	[1, 2, 3]	(s3, g1)	[2, 3, 1]
(s3, g2)	[2, 3, 1]	(s3, g3)	[3, 2, 1]
(g1, j1)	[3, 2, 1]	(g1, j2)	[2, 3, 1]
(g1, j3)	[3, 1, 2]	(g1, s1)	[1, 2, 3]
(g1, s2)	[3, 1, 2]	(g1, s3)	[2, 3, 1]
(g2, j1)	[2, 3, 1]	(g2, j2)	[2, 1, 3]
(g2, j3)	[1, 2, 3]	(g2, s1)	[2, 1, 3]
(g2, s2)	[3, 2, 1]	(g2, s3)	[2, 3, 1]
(g3, j1)	[3, 1, 2]	(g3, j2)	[1, 2, 3]
(g3, j3)	[1, 3, 2]	(g3, s1)	[1, 3, 2]
(g3, s2)	[3, 1, 2]	(g3, s3)	[3, 2, 1]

Table 5: Preferences for Pairs, the ranking is for the set that is unpaired yet

We can use these to look for instabilities in the final matchings, for the given set of preferences every possible final matching has an instability and hence there are no stable matchings for this set of preferences. We provide an algorithm to find a stable matching (when it exists) in the following section.

5.3 Finding a Stable matching if one exists

Now that we have shown that it is not necessary that a stable matching exists in all cases, we move our attention to trying to find one the matching if it does exist.

To look at this idea in higher dimensions it may be useful to understand the concept of k-partite graphs.

A graph $G = (V, E)$ is a k-partite graph if its vertex set V can be partitioned into k independent sets $V_1, V_2, \ldots, V_k$ such that for every edge $(u, v) \in E$, u and v belong to different sets V_i and V_j.

Now we can show that the problem of finding such a matching is computationally infeasible, we do this by showing that the Problem of determining whether given a graph, determining if it is 3-Partite or 3-Colorable is an NP-Complete problem. We show this via reduction from 3-SAT.

Given an instance of 3-SAT with variables $x_1, x_2, \ldots, x_n$ and clauses $C_1, C_2, \ldots, C_m$, we construct a 3-partite graph as follows:

1. For each variable x_i, we create three vertices in the graph representing the variable and its negation. Let's denote these vertices as x_i^0, x_i^+, and x_i^-, respectively.
2. For each clause C_j with literals l_1, l_2, l_3, we create a triangle (complete graph with three vertices) in the 3-partite graph. Each vertex of the triangle corresponds to one of the literals in the clause.
3. Connect each literal vertex to the corresponding variable vertices. For example, if the literal is x_i, connect it to x_i^+; if the literal is $\neg x_i$, connect it to x_i^-.

After constructing the 3-partite graph according to the reduction, we claim that the original 3-SAT instance is satisfiable if and only if the constructed graph is 3-partite.

- If the 3-SAT instance is satisfiable, we can assign colors to the vertices of the 3-partite graph such that no two adjacent vertices have the same color:
  - Color all variable vertices according to their truth values in the satisfying assignment.
  - Color all literal vertices according to the color of their corresponding variable vertices.

  Since each clause triangle contains exactly one vertex of each color, the graph is 3-partite.
- If the constructed graph is 3-partite, we can derive a satisfying assignment for the original 3-SAT instance:
  - Assign each variable x_i to true if its positive vertex is colored, and false otherwise.

  Since each clause triangle contains exactly one vertex of each color, it means that at least one literal in each clause is true under this assignment.

Therefore, determining if a graph is 3-partite is NP-complete, as it can be used to solve 3-SAT, a known NP-complete problem.

3D-Matching is nearly an equivalent problem to testing a graph for 3-partiteness, given some extra constraints.

Stable Matching represented as a Bipartite Graph problem

Oftentimes, bipartite graphs are useful to consider because we want to find a way to pair vertices from one part with vertices from another part. A **matching** is a collection of edges where no two edges share a vertex. In other words, a matching is a way of pairing up vertices that are connected. We will notate matchings by listing their edges. For example, matching vertex A with vertex C and vertex B with vertex D could be notated as AC, BD. A perfect matching is a matching where every vertex in the graph is included in some edge from the matching. A perfect matching is appropriate when we want to find a way to include every vertex in some pair.

In many cases, we want to do more than find just any matching; some matchings may be more desirable than others. This often happens when the vertices of a graph have preferences about who they will be matched with. For example, consider again the bipartite graph of people and jobs, where there is an edge connecting a person to a job when they are qualified for the job. Each person prefers some jobs over others; their preferences can be expressed as an ordered list, with their most preferred jobs at the top and their least preferred at the bottom. Each employer also has a similar list of potential employees. A bipartite graph that comes equipped with preference lists like this is called preference-labeled.

A **stable matching** in a preference-labeled bipartite graph is a matching such that there is no pair of vertices which prefer each other to their matched partners, and no vertex prefers an unmatched vertex to their matched partner.

The Gale-Shapely Algorithm in a graph algorithm form

- Choose one part of the graph to be the proposers.
- Each proposer proposes to the highest vertex on their preference list which has not rejected them.
- Each vertex being proposed to says "maybe" to their favorite of the vertices currently proposing to them, and rejects all other vertices currently proposing to them.
- Repeat steps 2 and 3 until you reach a step where nobody is rejected. At this point, you have a stable matching.

In a similar way any k dimensional matching is equivalent to finding pairings in a prefrence labelled k dimension-graph.

6 Our Algorithm

In section 3 we showed how to break up a preference list into groupwise and individual pairings. The idea is to use these pairings to evaluate any given random matching and save the comparison results so we do not have to perform the comparison each time.

Take the following final pairing:

$$(j_3, s_2, g_3), (j_1, s_1, g_2), (j_2, s_3, g_1)$$

If we use our preference lists to find that there are unstable groups (j_3, s_2, g_3) and (j_2, s_3, g_1) because (j_3, s_2) would rather be paired with g_1 instead and g_1 in turn also prefers them as its number 1 ranked group, then any matching that has this possibility can be instantly discarded so any pairing of the form:

$$(j_3, s_2, g_3), (-, -, -), (-, -, g_1)$$

can automatically be discarded. So we do this until we exhaust all possibilities and then if we cant find a instability we return that pairing as stable, otherwise we declare that no stable pairing exists for this set of preference lists.

Our code does a lot of pre-processing to convert the lists into desirable easy to process formats but the main pairing happens in the following function:

```python
def combine_rankings(list1, list2):
    combined_rankings = []

    pairs = list(product(enumerate(list1, start=1), enumerate(list2, start=1)))

    for (idx1, val1), (idx2, val2) in pairs:
        combined_score = idx1 + idx2
        combined_rankings.append((val1, val2, combined_score))

    sorted_rankings = sorted(combined_rankings, key=lambda x: x[2])
    converted_rankings = [f"({tup[0]}, {tup[1]})" for tup in sorted_rankings]
    return converted_rankings

for key in pref_:

    single_pref[key] = combine_rankings(pref_[key][0], pref_[key][1])

def matching3D(pairings, group_pref, single_pref):

  for res in pairings:

     groups = extract_groupings(res)

        all_pairs_1 = generate_pairs(groups[0])
        all_pairs_2 = generate_pairs(groups[1])
        all_pairs_3 = generate_pairs(groups[2])
```

```

       all = all_pairs_1 + all_pairs_2 + all_pairs_3

       stable = True

       for pair1 in all:

         for pair2 in all:

          if str(pair1[1]) + str(pair1[5]) == str(pair2[1]) + str(pair2[5])
          and pair1 != pair2:

            if group_pref[pair1[:8]][group_pref[pair1[:8]][:8].index(int(pair2[-2]))]
            < group_pref[pair1[:8]][group_pref[pair1[:8]][:8].index(int(pair1[-2]))]:

                if single_pref[pair2[-3:-1]].index(pair1[:8])
                < single_pref[pair2[-3:-1]].index(pair2[:8]):

                   stable = False
                   print(f"{pair1} and {pair2} form an instability\n")

                   break

        if stable == True:
            pairings[res] = True

    for key in pairings:

        if pairings[key] == True:

            print(f"{key} is a stable matching")
```

The entire code for the problem is available here. The code generates random preference lists and runs the algorithm on them, you can uncomment parts to add your own list too.

7 Further Work and Limitations

We have reviewed 3-dimensional matching in considerable detail and compared it to 1 and 2 dimensional counterparts and even came up with an algorithm that works considerably better than brute force. However, we did not try this for large preference lists of size above 5, this is because the data generated becomes very large, for 5 members from each set you have $5 \cdot 5 \cdot 5 \cdot 5 \cdot 5 = 3125$ different possible pairings and checking all of these takes a considerable time, even if we were to check it the possibility of an unstable pair goes higher and higher with a larger number of people to match and unless we have very specifically structured preference lists that are made to ensure that a stable match exists, we our not guaranteed to even get a single stable match out of those 3125. Although these results could be more formalized.

There may also a general way to extrapolate these results to higher order matchings but the complexity just increase exponentially as you increase the dimensions so analyzing higher dimensions becomes unneccesary.