The Stable Marriage Problem and Gale-Shapley Algorithm

Software Specifications and Correctness
Katrine Rachitsky
https://github.com/katkatkat9/StableMarriage
rachitk@mcmaster.ca
1306314

Table of contents

Background

Original problem

The stable marriage problem is a particular type of matching problem; one in which we begin with two sets of individuals (for the sake of explanation, we will assume the problem to be one of arranging marriages for the individuals), one set of suitors and one set of women. Each individual has a list which provides an order of preference for each person from the other set (each woman will begin with a list of men ordered by their preference for whom to marry, and each man will begin with a list of women ordered by their preference for who to marry). Herein lies the stable marriage problem: Given these two sets of individuals, as well as each individual's corresponding list, we must find a matching that is optimal. The matching should be optimal in that there exists no couple who prefer each other over their assigned partner (Kleinberg & Tardos, 2014). The solution to this problem is the Gale-Shapley algorithm, an algorithm developed by David Gale and Lloyd S. Shapley.

Contributors

Lloyd S. Shapley and David Gale worked together to come up with the Gale-Shapley algorithm, a well-known solution to the stable marriage problem. Shapley devoted himself to academia and applying the knowledge he gained from it; he is known for a wide variety of achievements, from being awarded a bronze star for cracking enemy codes during the second world war to coming up with what is known as the "Shapley value" (relevant in cooperative game theory, the "Shapley value" is essentially a way to determine payoff for each member of a team of players based on individual contributions) (Manlove & Wooldridge, 2013).

David Gale, on the other hand, is known for his creation of the 'top trading cycles' algorithm, an algorithm with applications in the academic and medical fields (choosing schools and assigning the kidneys of donors to patients in need of transplant). Gale is also a partial winner of the von Neumann Theory Prize, sharing the award with two others for their work which supplied the foundation of game theory, linear programming and non-linear programming (Manlove & Wooldridge, 2013).

Interestingly, only after Gale and Shapley had already published their Gale-Shapley algorithm had they learned that the same algorithm was already being applied in the medical field for 11 years (in particular, it was already being used to assign medical interns to respective residency positions throughout the United States) (Manlove & Wooldridge, 2013).

Applications

The National Intern Matching Program, developed by J. M. Stalnaker and F. J. Mullin, was a voluntary program used to solve the problem of assigning interns to residencies. At that point in time, the hospitals had more positions open than interns trying to secure them, which caused a myriad of problems (Manlove & Wooldridge, 2013). The system they had developed turned out to be functionally equivalent to the Gale-Shapley algorithm, if each hospital is considered a proposing suitor and each student a woman to propose to. The Gale-Shapley algorithm is otherwise used in many ways, including admissions to public schools (in which it was actually suggested to allow the applicants to be the 'proposers', since too many applicants were providing false preference data as a strategy to become assigned to their school of choice - this type of preference falsification and its possible advantages will be discussed in a later section of the report), admissions to daycare for children, recruitment of university faculty, assigning cadets in the military to respective branches, online dating /

match-making, and even auctioning off sponsored searches for search engines on the internet (Manlove & Wooldridge, 2013).

Properties

NP Classification of Stable Marriage

While Gale-Shapley is a cleverly written algorithm, and is still widely used today, it is not an algorithm which can be considered efficient in the traditional sense; this is because it is not an algorithm that solves a problem for which a solution can be computed within polynomial time, in fact, stable marriage is known to be an NP problem (Ronn, 1990). NP problems can be defined as the set of problems which can be solved by a non-deterministic algorithm as well as the set of all problems that have an efficient certifier (an algorithm which determines whether or not the problem has been solved given the output of the problem-solving algorithm) (Kleinberg & Tardos, 2014). One can see that a certifier could be written for the Gale-Shapley algorithm, as this would involve simply checking for stability by ensuring that for each matching there exists no other pair which prefers each other over their current mate. Additionally, one can see that the stable marriage problem can be solved by a non-deterministic algorithm which guesses a matching and verifies stability afterwards (Ronn, 1990).

Pseudocode

The following code snippet is the pseudo code that was used as a guide to implement the Gale-Shapley algorithm in Dafny (Kleinberg & Tardos, 2014).

```
Initially all men in set of men and w in set of women are un-engaged
While there is a man m who is free and hasn't proposed to every woman
      Choose such a man m
      Let w be the highest-ranked woman in m's preference list to whom m has
not yet proposed
      If w is free then
            (m, w) become engaged
      Else w is currently engaged to m'
            If w prefers m' to m then
                  m remains free
            Else w prefers m to m'
                  (m,w) become engaged, m' becomes free
      Endif Endif
Endwhile
Return the set of engaged pairs
```

Correctness Proof of The Gale-Shapley Algorithm

Claim: The Gale-Shapley algorithm has a runtime of O(n^2).
(Kleinberg & Tardos, 2014)

We must prove that each iteration of the algorithm comes closer to termination, or, in terms of the invariance theorem, we need to provide a bound function related to the loop guard

that becomes smaller as the iterations execute, eventually negating the loop guard and letting the program exit the loop.
In terms of this algorithm, within each iteration, a man proposes to a woman on his preference list. We can introduce a variable t, denoting the iterations, such that P(t) denotes the pairs (m, w) that have been attempted as matches by the end of iteration t. It follows that, at each successive iteration, the set P(t) for all attempted pairs grows: P(t) < P(t+1). Now keep in mind that only n^2 possible pairs exist, given n men and n women, and therefore the amount of attempted pairs in P can increase no more than n^2 times throughout execution of the algorithm. Note that progress in this algorithm cannot be measured by the number of engaged or un-engaged men or women, as this number could remain the same throughout multiple iterations and thereby could not be used as a bound function.

Claim: For each unengaged man found at any given point throughout execution of the Gale-Shapley algorithm, there is an unengaged woman.
(Kleinberg & Tardos, 2014)

Assume, at some point through execution of the loop, there is a man m who is not engaged but has already proposed to each woman on his preferences list. This implies that each woman on the list is engaged already, because the algorithm stipulates that a woman says yes to a proposal so long as she is not engaged already. For all women on the preference list to be engaged (and n women must exist on the preference list), there must be n men that they are engaged to, because each engagement is a pairing. This forms a contradiction, because there can only be a total of n men, so there cannot exist such a man within the set of n men.

Claim: The set of engaged couples generated by the time the Gale-Shapley algorithm terminates is a perfect matching: Each individual provided must be matched to another individual from the opposing set.
(Kleinberg & Tardos, 2014)

Assume that we have terminated, but are left with a man m who remains un-matched. In order to terminate, the Gale-Shapley algorithm must have iterated over every woman in m's preference list, and throughout every iteration the woman must have already been engaged. The previous proof indicates that this is a contradiction, because n women cannot already be engaged when there exists a man m in the set of n men.

Claim: The Gale-Shapley algorithm returns a set of pairs which constitutes a stable matching.
(Kleinberg & Tardos, 2014)

As we can already trust from the previous proof that our matching is perfect (each individual will be matched by the time the algorithm terminates), we assume an unstable pair that exists in the set of pairs returned. To have an unstable match, there must exist a couple (m, w) and another couple (m', w') such that one member of the first pair and one of the second pair prefer each other over their current match (for example, m prefers w' over w and w' prefers m over m'). In order for m to have ended up engaged to w, m's last proposal must have been to w. There are two possibilities here (and both will be proven impossible):

1. m had not proposed to w' yet
   herein lies a contradiction, as this would indicate that w is higher on m's preference list than w', meaning m prefers w over w'

2. m had proposed to w’ and w’ had rejected the offer because she preferred her current mate herein lies a contradiction, as this would indicate that w’ was already engaged and preferred her current partner over m, meaning w’ prefers m’ over m

Therefore, the set returned by the Gale-Shapley algorithm contains only stable matches.

Lying Women in Gale-Shapley

It is a known fact that the Gale-Shapley algorithm provides the most favourable matching to the ‘proposing’ set of individuals, which are, in the traditional example of the problem, the men. It is the preference lists of the men which get iterated through, starting from the woman the man prefers most, compared to all women that he hasn’t proposed to yet, while the women’s preferences only get taken into account once a new man has proposed to an already engaged woman (in this case she is given the choice, between the two men, of which she prefers). This inequality produces an ability to potentially improve their matching by falsifying their preference lists (Gonczarowski & Friedgut, 2011).

One must also note that any woman who has improved her matching by lying on her preference list (improved meaning her final match is ultimately a man who is higher on her preference list than the man she would have been matched with, had she been honest) has hindered the match of a man. One can convince themselves that this is true by assuming both the man and woman have improved their match as a result of the woman’s lie: If this was true, the original matching, for which the woman would have been worse-off, could not have been stable. This would mean that the man and the woman both prefer each other over whatever original match would have been generated for them, had the woman not lied - this would indicate an unstable match (Gonczarowski & Friedgut, 2011).

Indifference & its Impact on Gale-Shapley

The Gale-Shapley algorithm is relatively simple when strictly ordered preference lists are enforced, such that each individual on a preference list must have a distinct ranking from one another, for each list. When one considers, however, a list which allows for indifference between two or more individuals on the preference list, there are repercussions. In terms of producing a program that implements the algorithm, new issues arise. One must not only receive and store preference lists for each individual provided, but also maintain some explicit ranking for each individual on each preference list (instead of assuming an implicit one, based purely on the order of each of the given lists). This requires the chosen data structure to store much more information for each individual, and also requires the ability to easily access this data throughout execution of the algorithm. Note that the implementation provided in the code walkthrough does not handle indifference; this is mainly due to a combination of the reasons above, as well as the disadvantages associated with the verification tool that was chosen. These disadvantages are outlined in more detail in the description of the tool that was chosen to implement Gale-Shapley.

The most prominent impact indifference has on the output (as well as the path the code takes to generate the output) is the new levels of stability it introduces. The Gale-Shapley algorithm without indifference provides a stable matching, an important point that has come up many times in prior sections; however, the meaning of stability itself changes when indifference is introduced. The prior definition no longer sufficiently covers all possible cases, it only stipulates that for each pair (m, w) there does not exist a pair (m’, w’) such that one individual from each pair prefers each other over their current mate. For rankings which allow no duplicates in preference lists (no indifference, as indifference implies a ranking can be repeated - two individuals could be equally preferred), Gale-Shapley ensures that the pair stated above cannot exist. This is the only possible definition of stability when indifference is disallowed, but when indifference is allowed, we may have pairs that do not strictly prefer each other over their partners, but perhaps do not strictly prefer their partners over each other either. Our current definition of stability does not account for these cases, so we consider a more complete set of stability definitions with varying levels of strength (Irving, 1994).

Weak stability describes a matching in which there exists no couple such that each member prefers each other over their assigned partner. *Weak stability* is the same as our original definition of stability, without considering pairs that are indifferent between their current mate and a different partner. *Strong stability*, on the other hand, describes a matching in which there exists no couple (x, y) such that x prefers y to their assigned matching, and y either prefers x to their assigned matching or is indifferent between the two (the ranking of preferences for x and the assigned match are identical). Finally, *super stability* describes a matching in which there exists no couple such that each member of the couple either prefers or is indifferent between the other and their current partner (Irving, 1994).

Similar Problems

Similar algorithms

The stable-marriage problem itself belongs to a family of problems called stable matching problems. Generally, stable matching problems consist of groups of individuals, each with a list, indicating their preferences over other individuals for their match. The general goal of stable matching problems is to reach a state of stability within the matching; recall that stability and its strength depends on whether or not preference lists can contain indifferences between individuals. Some stable matching instances, when given the option for indifference, do not provide a stable matching at all (none exists). The most commonly known stable matching problems (other than the stable marriage problem, which has been covered up until now) include the roommate problem, the intern assignment problem, and the intern assignment problem with couples (Ronn, 1990).

The Roommate Problem

The roommate problem requires a single set of individuals, all needing to be matched to one another. Naturally, because there is only one set, the size of the set must be even so that each individual has a match by the time the program terminates (Ronn, 1990). Similar to the stable-marriage problem, each individual must provide a preference list to describe their preferences over their peers in the list. Instances of the roommate problem exist in which there are no stable matchings, regardless of whether or not ties are allowed. The stable marriage problem is really a special case of the roommate problem, one in which we must receive two different sets of individuals and ensure that each individual from one set is matched to one from the other (Ronn, 1990).

The Intern Assignment Problem

The intern assignment problem is essentially a permutation of the stable marriage problem, allowing any individual from one set to take on multiple partners from the other set. The algorithm which solves this problem is known as the National Internship Matching Program algorithm, and is regarded as a generalization of the Gale-Shapley algorithm (Ronn, 1990).

The Intern Assignment Problem with Couples

The intern assignment problem with couples is the most challenging, it is a far more complicated case to compute and does not guarantee a solution for many instances. This permutation of the intern assignment problem involves attempting to create a stable matching for couples on their internships, the purpose of including couples being that some may be married and would prefer to be assigned to internships that are geographically near each other (Ronn, 1990). Each member of the couple has their own, individual preference list which gets joined with the other member's. This joint preference list is then used to help determine which internships they can be matched to (Ronn, 1990). Instability is more complex to define for this particular problem, as there are three entities to satisfy. Instability for the intern assignment problem with couples can be defined in the following three cases; in one such case, there exists an intern that is not married, and there exists a hospital such that both the intern and the hospital prefer each other over their own matches (Ronn, 1990). Another case involves a hospital preferring an intern that is married and the intern preferring the hospital over their own matching, such that the joint assignment is overall improved (overall improved because neither member of couple suffers from a change of this nature, but one strictly benefits from it) (Ronn, 1990). The last case of instability involves two hospitals such that each prefers both members of one couple, with each member of the couple also preferring the corresponding hospital (Ronn, 1990).

Graphs and Gale-Shapley

The stable marriage problem can be generalized into a graphing problem, namely one in which we begin with two distinct sets of nodes (representing men and women) in a bipartite graph; as we proceed to try to find a stable matching, the problem becomes selecting start and end points for edges, given the criteria that the edge must begin in one node from one set and end in one node from the other, taking the preferences into account. Thus, a matching is essentially a set of disjoint pairs of individuals, connected by edges in a bipartite graph. To represent instability in the matching, we consider edges that 'block' each other (Cseh & Manlove, 2015). If there exists an edge mw that is not in the final matching, but either both m and w are unmatched or they both prefer one another over the partners assigned in the matching, mw is said to block the final matching. Therefore, in the case of bipartite graphs, we can refer to a stable matching as one that does not contain any blocking edges (Cseh & Manlove, 2015).

Tool chosen

Dafny

Dafny is a program verification tool, encompassing a programming language with specification constructs; Dafny strives to help the user write code that avoids inducing errors, performs the task that the user set out to, and is easy to write. The way in which Dafny helps with this is through the use of high level annotations which aid to prove code correctness. Using a high-level expression, the user can describe the desired behaviour for their code, and Dafny proves whether or not the code satisfies the high-level expression of the desired behaviour. This has a positive impact on software development, as it allows the developer to focus on posing requirements, writing code, and having the software system verify the correctness all the while.

There are many advantages to using Dafny for writing code as well as verifying its correctness, outlined below (Matias, n.d.):

- Dafny is simple
  - The programming and specification constructs available in Dafny are minimal, avoiding overwhelming the user with a wide variety of options, Dafny supplies only the basics to avoid complication
  - The constructs Dafny does provide include arithmetic, branching, assignment, strongly typed and weakly typed variable declarations, primitive data types, arrays, and user-defined data types
- Dafny is object oriented
  - Dafny allows users to create classes and generate data types themselves, by instantiating a class containing a constructor, data members, and methods
- Dafny provides verification and compiles code
  - Not all software verification tools provide a way to not only verify code but to execute it like a programming language; Dafny provides the benefits of both. Dafny compiles code and generates .NET framework byte code, making it possible to even execute the code in a .NET framework program
- Dafny uses and allows use of sets and sequences
  - Dafny provides use of unordered data structures using sets, and ordered data structures using sequences
- Termination can be proven implicitly with Dafny
  - Dafny is often able to determine the termination metric for while loops without any annotation, while in other cases the user is required to manually write up a decreases clause
- Dafny verifies quickly
  - Dafny has a relatively high speed for verifying a large amount of annotations. For example, a Dafny implementation of the Schorr-Waite algorithm includes thirty-two lines of loop invariants, a large amount of data to verify, but Dafny completes the process of reading and verifying the code in less than five seconds
- Dafny provides ‘forall’ loops
  - arrays and sequences often require removing particular members, accessing particular elements, or applying some operation on each element. Dafny supplies ‘forall’ loops, making most of the mentioned tasks take up no more than one line

Code Walkthrough

Method Declaration

We begin with the declaration of the method which takes men and women as input, and returns a stable matching of them as output. After some research on Dafny and its datatypes, maps were chosen to represent and store men, women and their preference lists. As such, the input for the matching method is a map referred to as men and another referred to as women. The key-value pairs of the map have types `int` and `array<int>`. Each integer key represents an individual (man or woman), while the array of integers, the associated values to the keys, represent the preference lists. The output returned is a map of integers, which represent the men and women who have been matched to each other.
We supply some requires clauses for the method to verify that the input we receive is acceptable considering the rules of the Gale-Shapley algorithm itself, as well as the conditions necessary for this particular implementation to work.

- `requires |men| != 0, requires |women| != 0`
  - Cardinality of the set of men and women must be non-zero
- `requires |men| == |women|`
  - Cardinality of men and women must be equal (requirement of Gale-Shapley, as the number of men and women must be equal for the matching to be perfect)
- `requires forall i :: 0 <= i < |men| ==> i in men && men[i] != null && men[i].Length == |women| && (forall j :: 0 <= j < men[i].Length ==> (men[i])[j] in women)`
  - Mapping of men must contain every number between 0 and the cardinality of the map (for example, if the amount of men is 10, the mapping should contain keys with all integers from 0-9 inclusive) using universal quantifier `forall i`. Additionally, the array of preferences for each key/individual `i` must be non-null and the length of each array must be equal to the cardinality of the mapping (another requirement of Gale-Shapley, because each preference list must contain each individual of the opposite sex, and thus must be equal to the cardinality of the mapping of individuals). Finally, for all valid array indices `j` of all arrays `men[i]`, `(men[i])[j]` must be in the given set of keys in the women mapping (meaning all elements of preference arrays must be an individual of the opposite sex's map).
- The next requirement stipulates, for women, the exact same conditions as for the prior requires clause
- Note the two 'commented out' ensures clauses, they were unfortunately not verifiable by Dafny, however they serve to prove that the matching returned to the user is a perfect matching by stating that, for all individuals within the men and women mappings, that those individuals exist as keys and values in the output mapping `matched_output`.

```
method matching(men: map<int, array<int>>, women: map<int, array<int>>)
returns (matched_output: map<int, int>)
  decreases *
  requires |men| != 0
  requires |women| != 0
  requires |men| == |women|
  requires forall i :: 0 <= i < |men| ==> i in men && men[i] != null &&
      men[i].Length == |women| && (forall j :: 0 <= j < men[i].Length ==>
      (men[i])[j] in women.Keys)
  requires forall i :: 0 <= i < |women| ==> i in women && women[i] != null
      && women[i].Length == |men| && (forall j :: 0 <= j < women[i].Length
      ==> (women[i])[j] in men.Keys)
  // ensures forall i :: 0 <= i < |men| ==> i in matched_output.Values;
  // ensures forall i :: 0 <= i < |women| ==> i in matched_output.Keys;
      {…}
```

Method Body

In the method body, we begin by initializing some variables for the loops. We instantiate a variable `matched` to be a mapping of integers to integers; this will hold our matching as it builds within the loop body. Next we instantiate a map called `indexLastAttempted`, a mapping of integers to integers. Unlike the `matched` mapping, we are not mapping members of the opposite sex to each other as key-value pairs; this map is meant to keep track of each man's latest proposal. In this way, we can avoid allowing any man to propose to any woman more than once in the inner loop body. This will be explained in further detail in the sections pertaining to loop body. The `indexLastAttempted` map is set to the value returned by the `initMen` method, which will be discussed further in the 'Additional Methods' section, but it suffices to know that this method takes the keys of `men` and creates a map in which each key, a man, is mapped to integer -1 (an out-of-bounds value which signifies that no woman has been proposed to yet by this man). The assertion following uses the ensures clause of the `initMen` method to verify that, for each integer `i` between 0 and the cardinality of the set of keys in the mapping of men, such that `i` is in men, `i` should also be a key in `indexLastAttempted` and the value associated with it should be -1.

Next we set a variable `currentMan` to be the integer result of a method called `getFreeMan` (discussed in more detail in the 'Additional Methods' section), which essentially finds a man that is in the set of keys from the men mapping, but not in the set of values of the matched mapping (and thereby not yet engaged). We then use the ensures clause of the `getFreeMan` method to assert that, if `currentMan` is out of the bounds of the set difference between the men and the engaged men, then for all `i` (representing men) between 0 and the cardinality of the aforementioned set difference, that man cannot be in the set difference (he is engaged, and more importantly, all men must be engaged). On the other hand, if `currentMan` is a valid index within the set difference of the men and the engaged men, this implies that `currentMan` is in the set difference. Our last initialization is the variable `currentPrefIndex`, which is an integer to be used within the loop bodies.

The '…' signifies the space in which the outer loop lies, which iterates over the un-engaged men to match them. There are two assertions (commented out) following the outer loop; one stating that, for all men from the given `men` mapping, that man is also in the values of the `matched` mapping. The next assertion states the same for women. These assertions reinforce that the matching is perfect. Once we have iterated over all unengaged men, we set the `matched_output` variable to the value of `matched`, and we return the `matched_output` variable with the complete, stable matching.

```
{
    var matched: map<int,int>;
    var indexLastAttempted: map<int,int> := initMen(men.Keys);
    assert (forall i :: 0 <= i < |men.Keys| && i in men ==> i in
      indexLastAttempted && indexLastAttempted[i] == -1);
    var currentMan: int := getFreeMan(men.Keys, matched.Values);
    assert ((currentMan == |men.Keys - matched.Values| ==> forall i :: 0 <=
      i < |men.Keys - matched.Values| ==> i !in (men.Keys - matched.Values))
      || (currentMan < |men.Keys - matched.Values| ==> currentMan in
      men.Keys - matched.Values));
    var currentPrefIndex: int;
      …
    // assert forall i :: 0 <= i < |men| && i in men ==> i in
      matched.Values;
    // assert forall i :: 0 <= i < |women| && i in women ==> i in
      matched.Keys;
    matched_output := matched;
    return matched_output;
}
```

Outer Loop

In the outer loop, we iterate through all unengaged men using the variable `currentMan`, and check that he is in the set difference between the men (`men.Keys`) and the engaged men (`matched.Values`). We supply the invariant which stipulates that the cardinality of the set difference between the men and engaged men is larger than or equal to 0, as this cardinality will get smaller over time and eventually reach a value of 0, at which point the loop is exited. Interestingly, this is not the bound function of the loop, because the number of engaged men does not necessarily rise steadily throughout the progression of the Gale-Shapley algorithm. Rather, the amount of women proposed to by each men rises, because no man proposes to the same woman once. Moreover, we are guaranteed that by the time a man has reached the end of his preference list, he will have been matched to someone. Although it is commented out, the decreases clause states that `|women| - indexLastAttempted[currentMan]`is the bound function of this loop (this clause has been commented out because Dafny had problems with confirming that `currentMan` was in `indexLastAttempted`, even though a prior assertion has been made to verify this fact, as well as some other blocking issues). This is reasonable, as each man's preference list is equal to the cardinality of the map of women, guaranteed by a requires clause, and because `indexLastAttempted[currentMan]` is guaranteed to be incremented at each iteration of the inner loop, since no man proposes more than once to the same woman.

Within the loop body, we set an array `preferences` to be the value associated with the key `currentMan` in men. Recall that each key in `men`, a man, has a value of an array, a preference list for that man. By setting `preferences` to the value corresponding to key `currentMan` in `men`, we set it to the preference array associated with the current man we are attempting to match. Now we make use of the `indexLastAttempted` map.

The name essentially explains the purpose, but recall that each man is initially paired with value -1, an invalid array index meant to signify that the man has not proposed to any women yet. If the current man is in the `indexLastAttempted` mapping (we already do know this to be the case from our prior assertion about `indexLastAttempted`, but Dafny requires us to state it regardless), and the value associated with key `currentMan` in `indexLastAttempted` is within the bounds of the preferences array, then we know that the man has already proposed to one or more women. If this condition passes, then we set `currentPrefIndex` to the value associated with the current man. This `currentPrefIndex` variable now points to the last index the current man reached in his preference array (meaning the current man most recently proposed to `preferences[currentPrefIndex]`). Now we can avoid repeating the last proposal by simply incrementing `currentPrefIndex.`

In the else block, we know the first conditional must not hold, meaning `indexLastAttempted[currentMan]` is -1. Since the current man has made no progress through his preferences array, we set `currentPrefIndex` to 0, so that we may begin looking at the preferences array at its first index (note that the ranking of preference for an individual in a preference list is the index itself, ordered from most preferred to least). At this point, the inner loop begins, explained in the next section. After the inner loop has completed, we have matched the current man with a woman, and can now move on to a new man. To do this, we set `currentMan` to the value returned by `getFreeMan`, and use the same assertion as in the method body.

Code block on next page…

```
while (currentMan in men.Keys - matched.Values)
     decreases *
     invariant 0 <= |men.Keys - matched.Values|
     // invariant -1 <= indexLastAttempted[currentMan] <= |women|
     // decreases |women| - indexLastAttempted[currentMan]
    {
        var preferences: array := men[currentMan];
        if (currentMan in indexLastAttempted && 0 <=
       indexLastAttempted[currentMan] < (preferences.Length - 1)){
               currentPrefIndex := indexLastAttempted[currentMan];
               currentPrefIndex := currentPrefIndex + 1;
       } else {
          currentPrefIndex := 0;
        }
        …
     currentMan := getFreeMan(men.Keys, matched.Values);
     assert ((currentMan == |men| ==> forall i :: 0 <= i < |men.Keys| ==> i !in men.Keys || i
       in matched.Values) || (currentMan < |men| ==> currentMan in men.Keys && currentMan !in
        matched.Values));
   }
```

Inner Loop

In the inner loop, we iterate through the preferences array (recall that the array's elements are ordered from most desirable match to least desirable match). In the outer loop, `currentPrefIndex` was set to act as a pointer towards the next index to use in the preferences array, and thus point towards the index of the next woman to be proposed to by the current man. The `currentPrefIndex` could have been set the following values:

- To 0, indicating the current man has never proposed to anyone, so he will propose to the very first woman on his index list
- To some other valid index in the preferences array, indicating that he has already proposed to women, so we can start where he left off instead of proposing to women that rejected him

Thus, while `currentPrefIndex` is still less than the length of the preferences array, we can continue proposing to new women on the list. We have an invariant, stipulating that `currentPrefIndex` is always between 0 and the length of the preferences array (inclusive), and we supply a clause that proves termination. The `decreases` clause stipulates that, through each iteration, the value representing the `currentPrefIndex` subtracted from the `preferences` array length is reduced. This acts as the bound function of our inner loop, and we decrease it by increasing the `currentPrefIndex` throughout each iteration. An additional, commented out invariant states that `indexLastAttempted[currentMan]` is always between -1 and the length of the preferences array (inclusive).

We begin the loop body by setting `indexLastAttempted`'s value at key `currentMan` to the `currentPrefIndex`, indicating that the current index in the preferences array we are accessing should not be attempted again, should this loop be entered again for the same `currentMan`. A new variable, `currentWoman` is set to the value of the preferences array at index `currentPrefIndex`. This is the woman that the current man will be proposing to, and now we enter the conditional block '…'.

```
while (currentPrefIndex < preferences.Length)
      invariant 0 <= currentPrefIndex <= preferences.Length
      decreases (preferences.Length - currentPrefIndex)
      // invariant -1 <= indexLastAttempted[currentMan] <= |women|
      {
      indexLastAttempted := indexLastAttempted[currentMan :=
      currentPrefIndex];
      var currentWoman: int := preferences[currentPrefIndex];
      …
      currentPrefIndex := currentPrefIndex + 1;
}
```

Inner Loop: Conditional

If the woman is already in the set of keys of the matched mapping (women are always the keys in the matched map, and men are always the values), then the woman is already engaged and the Gale-Shapley algorithm requires us to check whether or not she prefers the partner she is paired with (the value associated with her key in `matched`). Otherwise, we simply match her with the man proposing, as she has no better offers. This is the code in Gale-Shapley which allows us to end with a stable matching; if we only checked if a woman is engaged and never whether or not she prefers another man proposing, we would only get a perfect matching, not a stable one. The actual conditional checks if the woman is engaged by checking if she is in the `matched` map's set of keys (note that Dafny requires us to stipulate the `currentWoman` is in women already, despite the fact that we already have a requirement in the method declaration which ensures that every element of the preferences array has to be in the mapping of keys of the opposite sex). If she is not found to be in the `matched` map's set of keys, then we know the woman is single, and thus accepts the engagement to `currentMan`.

To do this, we set the `matched` mapping to become the matched mapping where key `currentWoman` is set with value `currentMan`. Now we simply break out of the loop, as there is no reason to continue iterating through the man's preference array if his match has already been found. If the woman already is engaged, we have to determine whether or not she prefers her current partner over the current man (if she does, then her current engagement will be broken and she will become engaged to `currentMan`, while her previous partner would be iterated over again in the outer loop to find him a new match). To find this out, we get the preferences array of the current woman, as well as the integer identifying the current man she is matched with (by accessing the value at `currentWoman`'s key in the `matched` mapping). In order to find which man she prefers out of the two, we find the indices at which they appear in her preferences array. We use a call to the `Find` method, explained further in the 'Additional Methods' section, to get the indices at which the current man and the current match lie in her preference list. We now use one more conditional to compare the indices of the men, and if the index of the current man is smaller than that of the current match assigned to the current woman, then he is higher up on her preference list; this means that the engagement between the woman and the current match can be broken, and that the woman can be matched with the current man because she prefers him. We break the engagement by setting the `matched` mapping to become the map with pairs `i` such that `i` is in `matched` and `i` is not the current woman, which means that the `matched` mapping takes out the key-value pair with key `currentWoman` to remove her previous engagement. Then we set matched to become the mapping where key `currentWoman` is set to `currentMan`, and, again, we break out of the loop because we no longer need to iterate through the preferences array once the man has become engaged. Recall from the previous section that immediately after these conditionals have been executed, the `currentPrefIndex` is incremented so that the man may move on to the next potential partner if he did not find a match in this iteration.

```
if (currentWoman !in matched.Keys) {
       matched := matched[currentWoman := currentMan];
      break;
} else if (currentWoman in matched.Keys && currentWoman in women) {
      var preferences: array := women[currentWoman];
      var man_matched: int := matched[currentWoman];
       var currentMan_index: int:= Find(preferences, currentMan);
      var man_matched_index: int := Find(preferences, man_matched);
if (currentMan_index < man_matched_index) {
       matched := map i | i in matched && i != currentWoman ::
matched[i];
            matched := matched[currentWoman := currentMan];
break;
      }
}
```

Additional Methods

getFreeMan

`getFreeMan` is a method which takes the set of men and engaged men as input and returns the first index at which there occurs a man who is not engaged; it is used for each iteration of the Dafny implementation to find the `currentMan` whose next partner will be found in the loop body. It begins with variable `index` initialized to 0, and, within the loop, searches through all indices of men. If an index is found to be in the set difference between men and engaged men, which indicates an unengaged man, the index is returned. If the index is not in the set difference, then we increment the index and try again for the next man.

The loop has a few annotations which allow us to prove the ensures clauses of the method. Firstly, we specify a bound function with a decreases clause, which specifies that the index subtracted from the cardinality of the set of men on each iteration will become smaller and eventually reach zero while we increment the index. The index begins at zero, and ends when it reaches the cardinality of `men`, so a reasonable loop invariant states that `index` is always between 0 and the cardinality of `men` (inclusive). Finally, our last invariant states that for any `i` such that it is smaller than the index we have currently reached within the loop, that `i` has already been determined to not be in the set difference of `men` and `engagedMen` (and thus, any man `i` that was tested before `index` was found to be engaged).

Given these annotations, we are able to prove the `ensures` clauses located with the method declaration. Firstly, the method requires a nonempty set of men, indicated by the requires clause which stipulates that the cardinality of the set of men must be larger than zero. Then we specify that for any valid index (`index < |men-engagedMen|`), the index is in the set difference between `men` and `engagedMen`; the next clause states that, if the index has reached the cardinality of the set difference, this means that all of the men were checked and determined to be engaged. Thus we can state that, for all `i` which are within the bounds of the set difference, `i` is not in the set difference (meaning there are no unengaged men left, and the outer loop may exit).

```
method getFreeMan(men: set<int>, engagedMen: set <int>) returns (index: int)
  requires |men| > 0;
  ensures index < |men - engagedMen| ==> index in men - engagedMen;
  ensures index == |men - engagedMen| ==> forall i :: 0 <= i < |men -
engagedMen| ==> i !in (men - engagedMen); // if index is larger than or
equal to cardinality of men, all i from 0 to cardinality of men must either
be in engagedMen or not in men at all
  {
  index := 0;
  while (index < |men|)
  decreases |men| - index
  invariant 0 <= index <= |men|;
  invariant forall i :: 0 <= i < index ==> i !in (men - engagedMen)
  {
    if (index in (men - engagedMen))
    {
      return index;
    } else {
      index := index + 1;
    }
  }
}
```

initMen

initMen is a simple method meant to create a mapping of integer keys to integer values, and is used to provide an indexLastAttempted map for the loop body to make use of. Each man should be initialized to an 'out of bounds' value (-1), to signify the man has not attempted to propose to any women on his preference list yet. This is done by using an index to iterate through the men with a while loop, and using preferenceCounter, our return variable, to set the value of the index (a man in the set of men) as a key, with value -1. The index is then incremented for the next iteration.

There are two invariants in the loop which help prove the ensures clause, the first stating that the index must be between zero and the cardinality of the set of men (inclusive). The other invariant states that for any i smaller than the current index (but larger than or equal to 0), i is in both the return variable (preferenceCounter) and the given set men, and lastly that the value associated with every such i is -1. This proves that for any i that has already been iterated over, the value at key i has already been set to -1, which helps Dafny prove our ensures clause.

The requires clauses of initMen simply stipulate that the input must be a set of men with a cardinality larger than 0, and that for any value between 0 and the cardinality of the set, that value should be in the set. The ensures clause wraps up the purpose of the method itself, stating that all elements of the set of men are in preferenceCounter as keys and that the value at each of these keys is -1.

```
method initMen(men: set<int>) returns (preferenceCounter: map<int,int>)
  requires |men| > 0
  requires forall i :: 0 <= i < |men| ==> i in men
  ensures forall i :: 0 <= i < |men| ==> i in preferenceCounter && i in men
&& preferenceCounter[i] == -1
{
  var index := 0;
  while (index < |men|)
  invariant 0 <= index <= |men|
  invariant forall i :: 0 <= i < index ==> i in preferenceCounter && i in
      men && preferenceCounter[i] == -1
  {
    preferenceCounter := preferenceCounter[index := -1];
    index := index + 1;
  }
}
```

Find

The find method outlined below serves the purpose of finding an element in an array, and returning the index of the element (used to compare the indices of men in a woman's preference array). If the element is found to not exist within the array, an index of -1 is provided to indicate no such index exists. Dafny needs us to stipulate what is true at each iteration of the loop, and then prove this so that it can prove that the end result, specified by the `ensures` clause, is true at termination. First we specify the easiest invariant, the one in which we state that the index variable is always between or equal to 0 and the length of the array being iterated through. Then we add that, for all possible indices `i` such that `i` is between 0 and the current index (or equal to 0), `a[i]` must not be the key (which is true, because, if we had iterated past the index in the array whose value holds the key, we would have returned already and broken out of the loop). This method was the only one not personally developed by me, it was found from the Dafny tutorial website and was integrated into the implementation <https://rise4fun.com/Dafny/>.

Given these two invariants, at termination we can safely say that, if the index is larger than or equal to 0, this indicates that the index is within the bounds of the input array and the array, at that index, holds the value of the given key. On the other hand, the next `ensures` clause states that a negative index implies that for all elements of the array, none of them match the given key. Looking at the very last assignment in the code, it is evident that this is the case, because the index is set to -1. This means that, if we did not return the index at some point in the loop body, we know that the key does not exist in the array at all, so we return -1 as an indication that the index for the key we are trying to find does not exist.

```
method Find(a: array<int>, key: int) returns (index: int)
   requires a != null
   ensures 0 <= index ==> index < a.Length && a[index] == key
   ensures index < 0 ==> forall k :: 0 <= k < a.Length ==> a[k] != key
{
   index := 0;
   while index < a.Length
      invariant 0 <= index <= a.Length
      invariant forall k :: 0 <= k < index ==> a[k] != key
   {
      if a[index] == key { return; }
      index := index + 1;
   }
   index := -1;
}
```

Test Cases

The following code snippets were used as test cases in the main method of the Dafny program to verify the correctness of the output provided, with their corresponding output following. Note that Dafny's array definitions require the developer to provide the size and type of the array first, then, one by one, provide entries for each index. In order to test, a preference array has to be created for each man and each woman, with each entry specifying a member of the opposite sex they could be matched with. From these arrays, we can now construct the mappings of men and women to their respective preference lists (for example, the man '0' has a preference array `man0`, thus key 0 is associated with the array `man0` in the mapping). The solution has already been pre-determined for all test cases and was verified against the output provided by the `matching` method to ensure the resulting matches are stable and perfect.

Test Case:

```
/*test 1*/

  var man0, man1, man2 := new int[3], new int[3], new int[3] ;
  man0[0], man0[1], man0[2] := 0, 1, 2;
  man1[0], man1[1], man1[2] := 1, 0, 2;
  man2[0], man2[1], man2[2] := 0, 1, 2;
  var woman0, woman1, woman2 := new int[3], new int[3], new int[3] ;
  woman0[0], woman0[1], woman0[2] := 1, 0, 2;
  woman1[0], woman1[1], woman1[2] := 0, 1, 2;
  woman2[0], woman2[1], woman2[2] := 0, 1, 2;

  var men: map<int, array<int>> := map[0 := man0, 1 := man1, 2 := man2];
  var women: map<int, array<int>> := map[0 := woman0, 1 := woman1, 2:=
      woman2];
  var results := matching(men, women);

  var i := 0;
  print "\n final matches are: \n";
  while (i < |results|) && i in results {
     print " woman ", i, " man ", results[i], "\n";
    i := i + 1;
  } /* solution is man 0, woman 0, man 1 woman 1 man 2 woman 2 */
```

Output:

```
Katrines-MacBook-Pro:~ katrine$ wine galeshapley.exe

 final matches are:
 woman 0 man 0
 woman 1 man 1
 woman 2 man 2
```

Test Case:

```
  /*test 2*/

  var man0, man1, man2, man3 := new int[4], new int[4], new int[4], new
int[4] ;
  man0[0], man0[1], man0[2], man0[3] := 2,3,1,0;
  man1[0], man1[1], man1[2], man1[3] := 3,2,0,1;
  man2[0], man2[1], man2[2], man2[3] := 0,2,1,3;
  man3[0], man3[1], man3[2], man3[3] := 1,3,0,2;

  var woman0, woman1, woman2, woman3 := new int[4], new int[4], new int[4],
new int[4] ;
  woman0[0], woman0[1], woman0[2], woman0[3] := 3,0,1,2;
  woman1[0], woman1[1], woman1[2], woman1[3] := 2,1,0,3;
  woman2[0], woman2[1], woman2[2], woman2[3] := 2,1,0,3;
  woman3[0], woman3[1], woman3[2], woman3[3] := 3,0,1,2;

  var men: map<int, array<int>> := map[0 := man0, 1 := man1, 2 := man2, 3 :=
man3];
  var women: map<int, array<int>> := map[0 := woman0, 1 := woman1, 2:=
woman2, 3 := woman3];
  var results := matching(men, women);
  var i := 0;
  print "\n final matches are: \n";
  while (i < |results|) && i in results {
     print " woman ", i, " man ", results[i], "\n";
    i := i + 1;
  } /*solution is woman 0 man 2, woman 1 man 3, woman 2 man 0, woman 3 man
1*/
```

Output:

```
Katrines-MacBook-Pro:~ katrine$ wine galeshapley.exe

 final matches are:
 woman 0 man 2
 woman 1 man 3
 woman 2 man 0
 woman 3 man 1
```

Test Case:

```
/*test 3*/

  var man0, man1, man2 := new int[3], new int[3], new int[3] ;
  man0[0], man0[1], man0[2] := 0, 1, 2;
  man1[0], man1[1], man1[2] := 0,2,1;
  man2[0], man2[1], man2[2] := 1,2,0;
  var woman0, woman1, woman2 := new int[3], new int[3], new int[3] ;
  woman0[0], woman0[1], woman0[2] := 1, 0, 2;
  woman1[0], woman1[1], woman1[2] := 2,1,0;
  woman2[0], woman2[1], woman2[2] := 1,0,2;

  var men: map<int, array<int>> := map[0 := man0, 1 := man1, 2 := man2];
  var women: map<int, array<int>> := map[0 := woman0, 1 := woman1, 2:=
woman2];
  var results := matching(men, women);

  var i := 0;
  print "\n final matches are: \n";
  while (i < |results|) && i in results {
    print " woman ", i, " man ", results[i], "\n";
    i := i + 1;
  } /* solution is  woman 0 man 1, woman 1 man 2, woman 2 man 0*/
```

Output:

```
Katrines-MacBook-Pro:~ katrine$ wine galeshapley.exe

 final matches are:
 woman 0 man 1
 woman 1 man 2
 woman 2 man 0
```

Running galeshapley.dfy

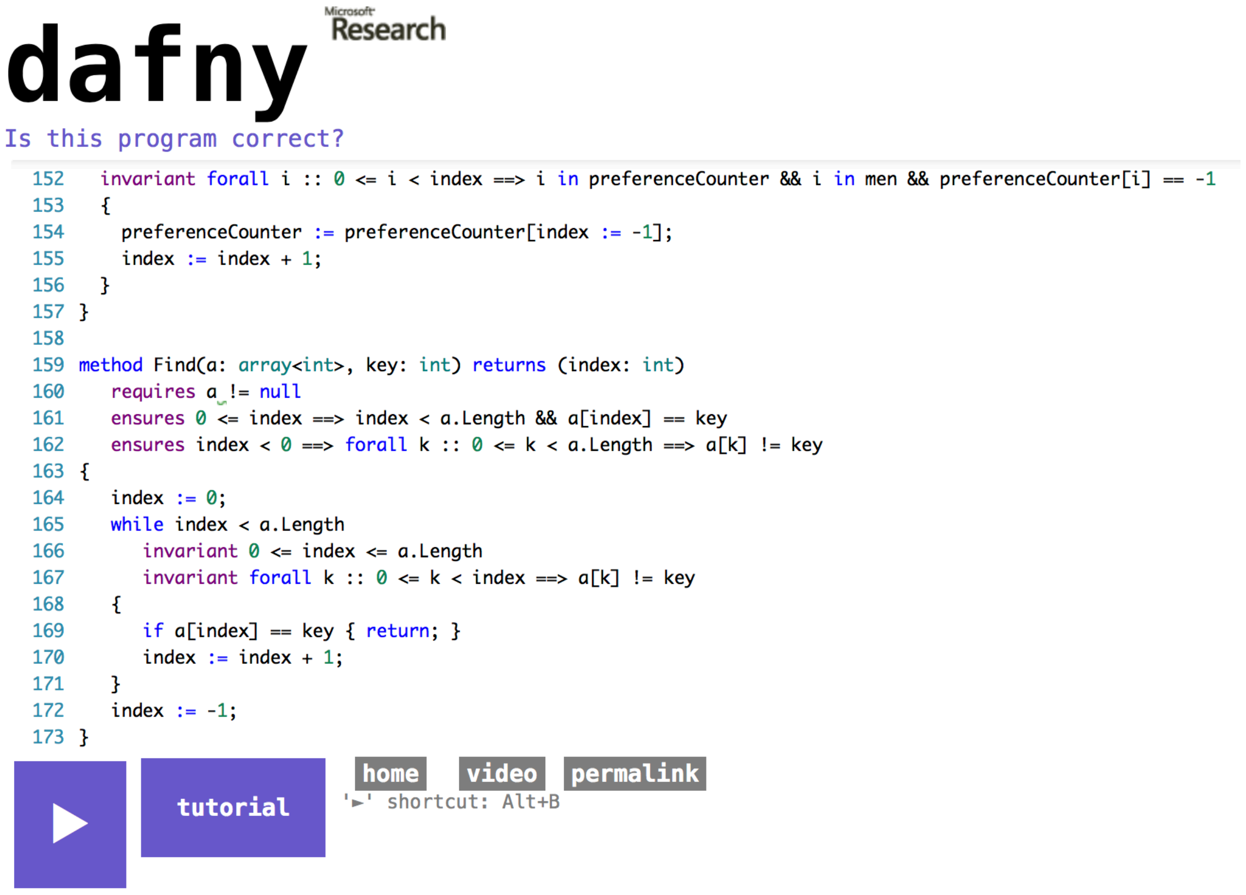
The implementation is provided in file *galeshapley.dfy*, and can be run:

- On the website
  - Online user interface
- Locally
  - Windows
    - Visual studio with Dafny extension
  - Mac and Linux
    - Dafny and wine

Website

The Dafny website https://rise4fun.com/Dafny/ provides an online user interface, this is the easiest way to run *galeshapley.dfy* by far because it requires no installation. The contents of *galeshapley.dfy* can simply be pasted into the user interface. To compile and run, the user must press the large ‘play’ button. After a few seconds pass (for compilation, validation and execution), the output is displayed to the user.

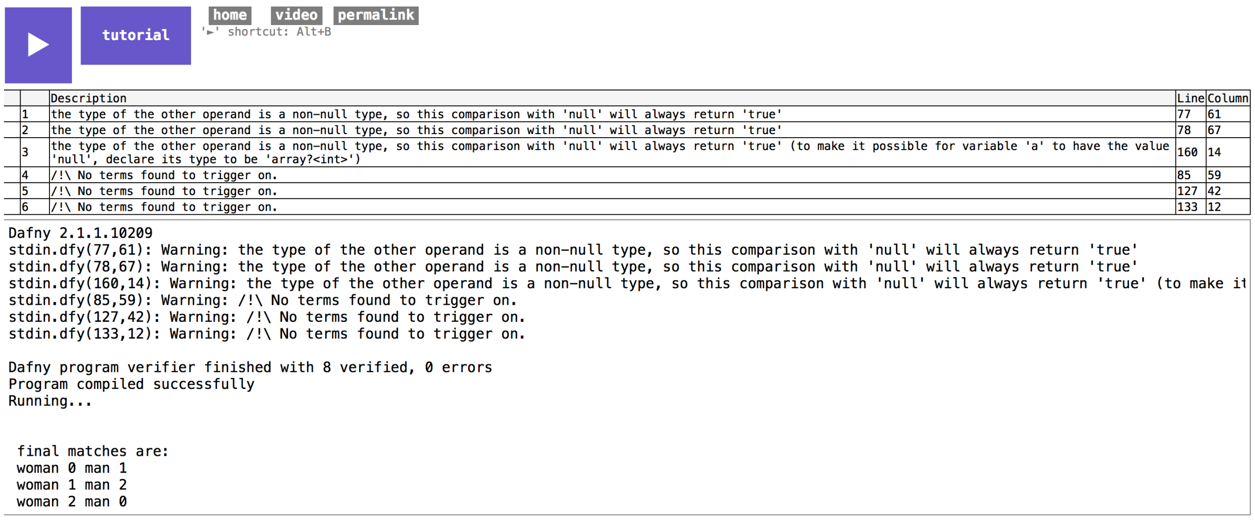
Figure 1: Dafny website’s user interface, with *galeshapley.dfy*’s contents pasted into text area



```
    invariant forall i :: 0 <= i < index ==> i in preferenceCounter && i in men && preferenceCounter[i] == -1
    {
      preferenceCounter := preferenceCounter[index := -1];
      index := index + 1;
    }
  }

  method Find(a: array<int>, key: int) returns (index: int)
     requires a != null
     ensures 0 <= index ==> index < a.Length && a[index] == key
     ensures index < 0 ==> forall k :: 0 <= k < a.Length ==> a[k] != key
  {
     index := 0;
     while index < a.Length
        invariant 0 <= index <= a.Length
        invariant forall k :: 0 <= k < index ==> a[k] != key
     {
        if a[index] == key { return; }
        index := index + 1;
     }
     index := -1;
  }
```

Figure 2: Output from pressing ‘play’ button



	Description	Line	Column
1	the type of the other operand is a non-null type, so this comparison with 'null' will always return 'true'	77	61
2	the type of the other operand is a non-null type, so this comparison with 'null' will always return 'true'	78	67
3	the type of the other operand is a non-null type, so this comparison with 'null' will always return 'true' (to make it possible for variable 'a' to have the value 'null', declare its type to be 'array?<int>')	160	14
4	/!\ No terms found to trigger on.	85	59
5	/!\ No terms found to trigger on.	127	42
6	/!\ No terms found to trigger on.	133	12

```
Dafny 2.1.1.10209
stdin.dfy(77,61): Warning: the type of the other operand is a non-null type, so this comparison with 'null' will always return 'true'
stdin.dfy(78,67): Warning: the type of the other operand is a non-null type, so this comparison with 'null' will always return 'true'
stdin.dfy(160,14): Warning: the type of the other operand is a non-null type, so this comparison with 'null' will always return 'true' (to make it
stdin.dfy(85,59): Warning: /!\ No terms found to trigger on.
stdin.dfy(127,42): Warning: /!\ No terms found to trigger on.
stdin.dfy(133,12): Warning: /!\ No terms found to trigger on.

Dafny program verifier finished with 8 verified, 0 errors
Program compiled successfully
Running...


 final matches are:
 woman 0 man 1
 woman 1 man 2
 woman 2 man 0
```

Locally

Windows:

Required external dependencies:

- Visual Studio 2012 Ultimate
- Visual Studio 2012 sdk extension
- Code contract extension
- NUnit test adapter
- Lit

1. Clone source code.
    - Dafny - https://github.com/Microsoft/dafny
    - Boogie - https://github.com/boogie-org/boogie
    - BoogiePartners - https://github.com/boogie-org/boogie-partners
    - **copy** Coco.exe to \boogiepartners\CocoRdownload
      Coco.exe can be found in:
      http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/
2. Build the following projects in the following order:
    - boogie\Source\Boogie.sln
    - dafny\Source\Dafny.sln
    - dafny\Source\DafnyExtension.sln
3. Conventions:
    - Set “General:Tab” to “2 2”
    - For “C#:Formatting:NewLines Turn everything off except the first option.

<Official Installation Instruction for Windows>
https://github.com/Microsoft/dafny/wiki/INSTALL

Mac and Linux:

For compiling Dafny code:
Instructions found at https://www.cs.cmu.edu/~mfredrik/15414/hw/guide.pdf

- Find out if you have mono installed by executing command `mono -V` in terminal
  - If you do not have it installed, download it from the following link http://www.mono-project.com/docs/getting-started/install/mac/ and run the *.pkg* file
- Download the latest release for your operating system from https://github.com/Microsoft/dafny/releases (for example, the latest for Mac is currently a file named *dafny-2.1.1.10209-x64-osx-10.11.6.zip*)
- Unzip the file and place the directory in the preferred location on your machine
- Move to the dafny directory you have just unzipped, and execute command `mono Dafny.exe` (should yield output '`*** Error: No input files were specified.`')
- Create a file *test.dfy* with the following contents:

```
method Main() {
 print "hello, Dafny\n";
assert 1 < 2;
}
```

- Execute command `mono Dafny.exe test.dfy`
  - If this does not work, you may have to add Dafny to your path. Execute command `export PATH=/Library/Frameworks/Mono.framework/Commands:$PATH` and try again
- Note that Dafny code can generally be run with command `mono Dafny.exe <filename>.dfy`, but this needs to be done within the dafny directory downloaded. To run Dafny code outside of the dafny directory, you can alias dafny on your machine by adding the following line to *.bash_profile*:
  `alias dafny="mono <path to Dafny directory>/dafny/Dafny.exe"`
- After this these steps have been completed, you can now run *.dfy* files with command `dafny <filename>.dfy`

Example output from compiling Dafny code (*galeshapley.dfy*):

```
Dafny 2.1.1.10209
/Users/katrine/Documents/GitHub/StableMarriage/galeshapley.dfy(77,61):
Warning: the type of the other operand is a non-null type, so this
comparison with 'null' will always return 'true'
/Users/katrine/Documents/GitHub/StableMarriage/galeshapley.dfy(78,67):
Warning: the type of the other operand is a non-null type, so this
comparison with 'null' will always return 'true'
/Users/katrine/Documents/GitHub/StableMarriage/
galeshapley.dfy(160,14): Warning: the type of the other operand is a
non-null type, so this comparison with 'null' will always return
'true' (to make it possible for variable 'a' to have the value 'null',
declare its type to be 'array?<int>')
/Users/katrine/Documents/GitHub/StableMarriage/galeshapley.dfy(85,59):
Warning: /!\ No terms found to trigger on.
/Users/katrine/Documents/GitHub/StableMarriage/
galeshapley.dfy(127,42): Warning: /!\ No terms found to trigger on.
/Users/katrine/Documents/GitHub/StableMarriage/
galeshapley.dfy(133,12): Warning: /!\ No terms found to trigger on.

Dafny program verifier finished with 8 verified, 0 errors
Compiled assembly into galeshapley.exe
```

Note that warnings are acceptable and do not indicate any issues with the verification of the code, but errors do. A quick way to check for issues is to inspect the final two lines of the output from compilation:

```
Dafny program verifier finished with 8 verified, 0 errors
Compiled assembly into galeshapley.exe
```

If the code verified with `x verified, 0 errors`, the code was deemed correct by the Z3 solver based on the verification conditions passed to it by Boogie.
Additionally, if the code has any errors, no executable file (*galeshapley.exe*) will be generated. Now we move on to how to execute the Dafny program.

For running executable Dafny code:
Instructions found at https://www.davidbaumgold.com/tutorials/wine-mac/#part-3:-install-wine-using-homebrew

- In terminal, execute command `brew install wine`
  - If this does not work, you may be missing a dependency (xquartz), so execute command `brew cask install xquartz`
  - When this installation has completed, try `brew install wine` again

To see example output from executing *galeshapley.exe*, see the output blocks in the 'Test Cases' section of this report.

Future Work

While the implementation does include a number of assertions, invariants, ensures clauses, and decreases clauses, among other annotations, there are still more aspects of Gale-Shapley which could be proven using Dafny (the commented out portions reveal many of these). Future additions to this implementation could include:

- Supplying a decreases clause for the outer loop, verifying termination of Gale-Shapley
- Supplying assertions after outer loop termination verifying that all men and all women are matched once the 'matching' method is complete

Termination in Gale-Shapley is ensured by the fact that each man only has a finite amount of women to propose to. Every man will be matched before they have reached the end of their preference list (see 'Correctness Proof of The Gale-Shapley Algorithm' section), and therefore the outer loop is bounded by how many women are left in every man's preference list. In terms of this specific implementation, `IndexLastAttempted[currentMan]` could be used, as it holds the latest index reached within the preferences list, and could be compared to the length of the preferences array for `currentMan`. This would act as a bound function for the outer loop.

A perfect matching could also be proven with assertions after the outer loop, or as an ensures clause, stating that all men and all women have been paired by the time the loop is finished. Since the outer loop's guard states that the loop should only continue while `currentMan in men.Keys - matched.Values`, meaning 'while there exists a man who is not yet engaged', this is surely possible (at least for men). This becomes difficult to do for this implementation, though, considering Dafny does not allow method calls within a loop guard. This means that, instead of calling `getFreeMan` within the loop guard and using the ensures clauses from that method to prove that, if the current man supplied is within bounds, he is unengaged (and engaged if he is not within bounds), we call it at the end of the method call. Unfortunately, while we can make an assertion after the method call regarding the output, we cannot use this data easily in the loop guard, and thus cannot make assertions about what must be true about the men once the loop has been exited.

All in all, this project provided a very different programming experience from the norm; instead of speeding towards a solution to the problem, one must consider every possible path the code could take before writing any statement when programming using a verification tool, and always take into consideration whether the code one is currently writing brings them closer to their goal or not. Correct by construction programming is a method which takes more time throughout development, but takes drastically less maintenance and testing, due to the fact that all code paths have already been proven to exhibit the desired behaviour; otherwise, the verifier would have warned the developer of an error long before the testing phase was even reached. Programming using a verification tool forces the developer to not only look at the specific section of code they are working on, but to plan and develop the algorithm as a whole, so that each piece is meticulously outlined before a single line is written. Sometimes annotations need to be re-written to verify the behaviour in different ways as changes are made to the code, but the overall goal always remains the same, as the developer always works to satisfy the constraints they defined as they began development; this results in code that is not only assumed to be correct after passing a few simple test cases, but verified to be correct given the conditions defined by the developer themselves.

References

Cseh, & Manlove, D. F. (2015). Stable Marriage and Roommates Problems with Restricted Edges: Complexity and Approximability. *Algorithmic Game Theory Lecture Notes in Computer Science,* 15-26. doi:10.1007/978-3-662-48433-3_2.

Gonczarowski, Y. A., & Friedgut, E. (2011). On Sisterhood in the Gale-Shapley Matching Algorithm (Unpublished master's thesis). Cornell University.

Harrenstein, P., Manlove, D., & Wooldridge, M. (2013). The Joy of Matching. *AI and Game Theory,* 81-85.

Irving, R. W. (1994). Stable marriage and indifference. *Discrete Applied Mathematics, 48*(3), 261-272. doi:10.1016/0166-218x(92)00179-p

Kleinberg, J., & Tardos, É. (2014). *Algorithm design*. Harlow: Pearson.

Matias, M. J. (n.d.). Program verification of FreeRTOS using Microsoft Dafny (Unpublished master's thesis).

(n.d.). Retrieved February 12, 2018, from https://rise4fun.com/Dafny/

Ronn, E. (1990). NP-complete stable matching problems. *Journal of Algorithms, 11*(2), 285-304. doi:10.1016/0196-6774(90)90007-2.