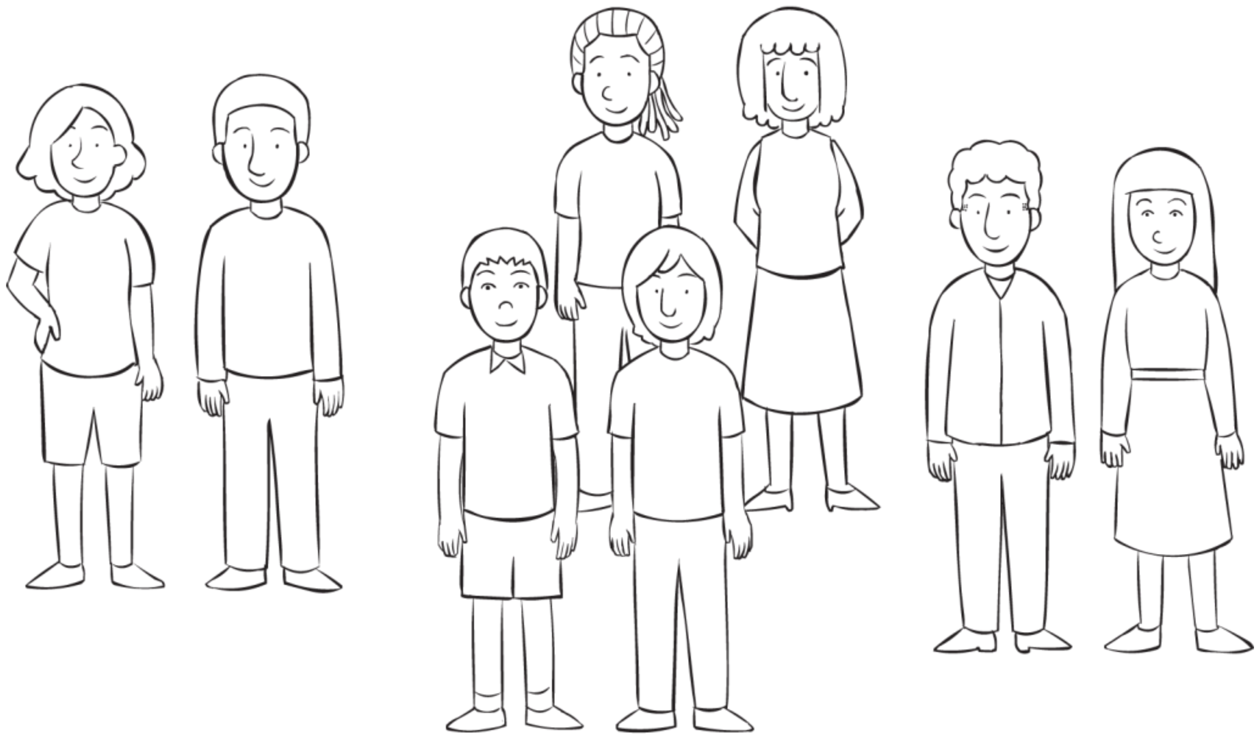


The Gale–Shapley algorithm

(deferred acceptance or propose-and-reject algorithm)

Luke Hazelton

CSCI411

4/6/2023

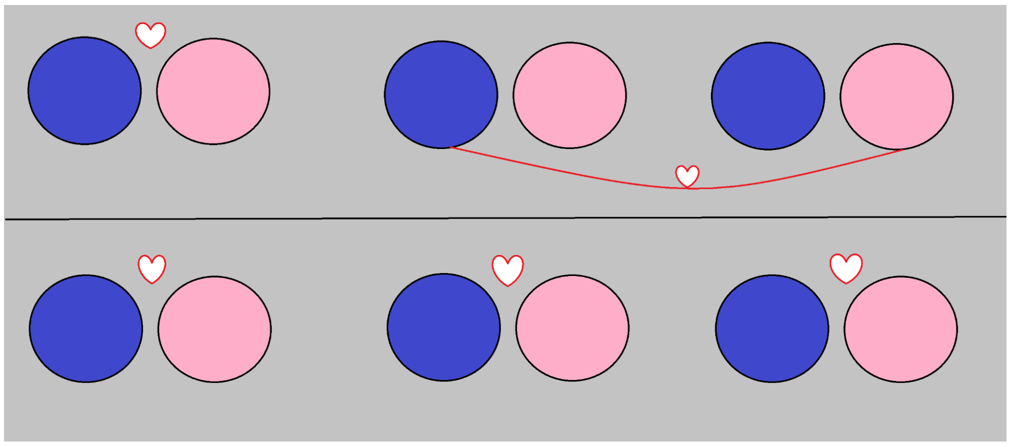
Introduction

History

In the 1950's David Gale and Lloyd Shapley proved that for any equal number of participants in two lists, it is always possible to find a matching in which every pair is stable, and presented an algorithm that would do so. This algorithm is used to solve what is known as a stable marriage problem, or SMP. The Gale-Shapley algorithm would first be put to practical use in the 1950's in the National Resident matching program, which was a non-profit government program created to place medical students into residency training programs within the US. This program is still up and running today helping medical students match with a hospital that is best for them. Since its creation this algorithm has had many uses for pairing different kinds of things like: employees and employers, schools and applicants, organ donors and those that need them, college admissions and even couples in online dating.

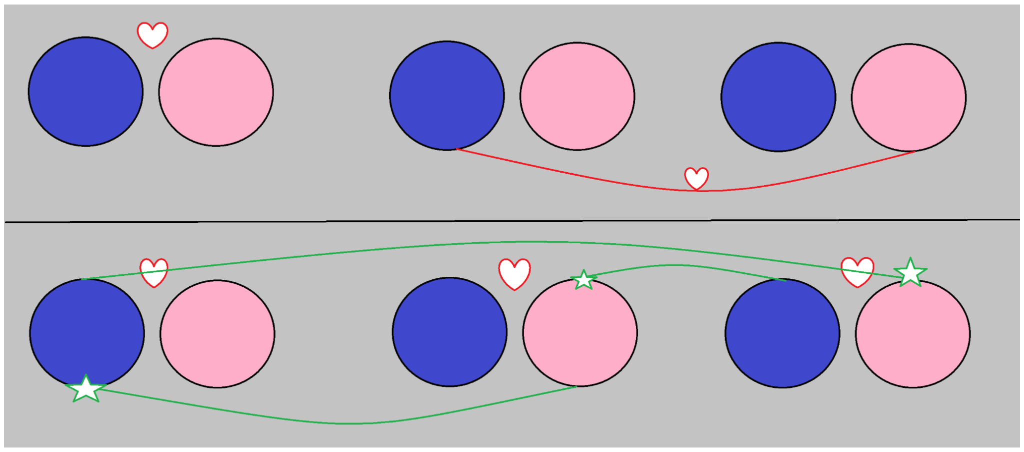
Explanation

The Gale-Shapley algorithm functions as a solution to the stable marriage problem. In order to explain this algorithm I will use the example of marriage, with proposers and acceptors who are both people. To define this algorithm we first need to define the problem that it is solving. Our desired solution to a stable marriage problem is a **set of n pairs made up of two sets of n people**, with one person from each of these sets appearing in each pair. This means that each individual can only be paired once, resulting in the pairs being a direct map between the two separate sets of people. This concept is simple enough, but the real meat of the algorithm comes from the individual preference that each person from each set has, to whom they would prefer to be matched with from the other set of people. Lastly the final set of matches needs to fulfill one more condition which is at the core of this algorithm ,which is that our set of n pairs needs to be comprised of matches in which there are no two people who prefer to be with each other over their current partners, thus creating an optimal stable match between every pair.



(top instance not a stable match, bottom is stable match)

Above depicts the most basic form of a non stable match, but I think this simple depiction could be too vague. For instance, if the man from pair two preferred the woman from pair three but the woman from pair three preferred her current match, the matches would still be stable. To more properly explain the SMP can go a step further by showing that even if the man from pair one preferred the woman in pair two, woman in pair two preferred the man in pair three, and the woman in pair three also preferred the man in pair one, this would still be considered a stable match. Even with only three people in each list, this can be tough to visualize so here is the same list as before, but with the second instance preferences clarified.


(top instance not a stable match, bottom is stable match)

The key idea here is reciprocation. Each green line indicates a one way preference, with the star being the point of origin. As you can see each pair of people can/will have different preferred matches, but a stable match will have these pairs organized in such a way that each preferred match outside of their pair is not reciprocated. To reiterate, what the Gale-Shapley algorithm is really doing is arranging two lists of size n which have a list of preferred matches to “people” in the respective opposite list, and arranges both into a list of pairs size n such that each person in each pair will not **mutually prefer** someone outside of their pair.

Intuition

First an optimal substructure for this algorithm is that of an array containing a list of ids or names of the person from the opposite list matching to the current list. Next to explain the intuition for this algorithm I will use what I think is the most important use case of the algorithm, which pairs organ donors to organ acceptors, this will still involve the lists still being composed of people. Next we’re going to need the preference lists for each list of people. The structure of these preference lists will be that of a 2D array, with each array representing a person, and the list inside of that array being composed of donors/acceptors ranked in order of compatibility/preference to their indexed person. In initializing this algorithm we will need each person in both the donor and acceptor matching list to be flagged as unmatched. Next we will iterate through the donor group to look for a suitable acceptor. As we iterate through the donor list, donors who have not been matched with an acceptor already will propose to their most preferable match from their matching list that has not already rejected the offer. The acceptor will then accept or reject this proposal based on a few criteria: If this is the acceptors first offer, they will of course accept it as they have no other option. Next if an acceptor already has a more preferable donar locked in, they will then reject this offer as they have no need for it. Finally, if an acceptor already has a previous donor locked in but the offer from the new donor is more compatible, they will accept the new donor and get rid of their previous donor. This pattern of matching will loop until all patients in both lists have been matched. After termination we will then be left with a list of people with each index corresponding to who they are matched with, for example: in list person_list acceptor 1 will be matched with the donor at person_list[1].

Pseudocode

I'm sure this isn't the most efficient approach possible, as every member of each list's preferred list has to be iterated over. If I were to fully optimize it, I believe another ordering function could lighten the load. The inputs for this algorithm will be in the main, and will consist of picking the size of the donor and acceptor lists while also populating both of the preferred lists. We will also pick the size of each preferred list as we populate. This will give us more control when testing GSA.

```
function GSA(acc_pref, donor_pref, acc_match, donor_match, acc_propose)
    n = size of acc_pref  // get the number of acceptors/donors
    for i from 0 to n-1:
        acc_match[i] = -1
        donor_match[i] = -1
        acc_propose[i] = 0  // start proposing with the first choice for each acceptor
    free_count = n
    while free_count > 0:  // while there exists a free patient (an acceptor without a donor)
        p = -1
        for i from 0 to n-1:
            if acc_match[i] == -1:
                p = i
                break
        for i from 0 to n-1:
            d = acc_pref[p][acc_propose[p]]  // get the current preferred donor for the current
acceptor
            if donor_match[d] == -1:  // if the donor is free, match the acceptor with the donor
                acc_match[p] = d
                donor_match[d] = p
                free_count -= 1
            else:  // if the donor is already matched
                p2 = donor_match[d]
                found_p = false
                found_p2 = false
                for j from 0 to n-1:
                    if donor_pref[d][j] == p:
                        found_p = true  // check if the current acceptor is preferred to the current match
                    if donor_pref[d][j] == p2:
                        found_p2 = true  // check if the current match is preferred to the current acceptor
                    if found_p and not found_p2:  // if the current acceptor is preferred to the current
match
                        acc_match[p] = d
                        donor_match[d] = p
                        acc_match[p2] = -1
                        break
                    else if found_p2:  // if the current match is preferred to the current acceptor
                        break
            acc_propose[p] += 1  // move on to the next preferred donor for the current acceptor
```

Runtime Analysis

Initializing and receiving my data will all take constant time, so the piece of my code that needs to be focused on is the while loop inside of GSA(). This while loop will run n times because at least one acceptor match is made each time it iterates, taking O(n) time. But inside this while loop is a for loop that iterates over each donor per the current acceptor also taking O(n) time. Thankfully, the second for loop is outside of the first. But this still means that the GSA algorithm will run in **O(n^2)** time, this is primarily because of the main while loop within the GSA function.

Addressing Feedback

Reviewer 1 Bryan Brooks: "I thought the introduction and nicely commented pseudocode explanation was effective and clear. Great explanation of why it is important."
Couldn't change much based on this one but I appreciate it nonetheless.

Reviewer 2 Dylan Hall: "I thought the backstory of the algorithm was informative. I thought the code was well commented for clarity. I wasn't quite sure what it was doing at a high level."

Feedback was good, because of this I attempted to more clearly explain a few different ways to understand the Gale-Shapley Algorithm. I even summarized and reiterated it at the end of my explanation section, so hopefully that helps clear things up.

Reviewer 3 Isaac Sanchez: "Interesting topic, I found the applications of the algorithm curious and intriguing in how it's used. The pseudocode was well documented making it easier to follow along. Runtime explanations were clear." Also couldn't change much based on this one but I still appreciate it nonetheless as well.

My Comments

Speaker: Dylan Hall
Reviewer 1 Luke Hazelton: "I liked your explanation of the pseudo code you provided, what are some of the practical applications that this algorithm is used for?"

Speaker: Isaac Sanchez - ID3
Reviewer 3 Luke Hazelton: "Overall the presentation was very effective on informing, but I was a little fuzzy on what the algorithm did at a low level."

Speaker: Bryan Brooks
Reviewer 1 Luke Hazelton: "Great visualizations provided. With the run-time being n! can you think of a way to cut down the runtime to be more efficient?"

Bibliography

Gale–Shapley algorithm (2023) *Wikipedia*. Wikimedia Foundation. Available at: https://en.wikipedia.org/wiki/Gale%E2%80%93Shapley_algorithm (Accessed: April 6, 2023).https://en.wikipedia.org/wiki/Gale%E2%80%93Shapley_algorithm

[Ryan Schachte]. (2017, January 17). *The Stable Matching Algorithm - Examples and Implementation* [Video]. Youtube. https://youtu.be/FhRf0j068ZA

Gale-Shapley algorithm (no date) *Encyclopædia Britannica*. Encyclopædia Britannica, inc. Available at: https://www.britannica.com/science/Gale-Shapley-algorithm (Accessed: April 6, 2023).https://www.britannica.com/science/Gale-Shapley-algorithm

detailed implementation of the gale-shapley algorithm - university at Buffalo (no date). Available at:

https://cse.buffalo.edu/faculty/atri/courses/331/fall14/handouts/GS-details.pdf
(Accessed: April
7,2023).https://cse.buffalo.edu/faculty/atri/courses/331/fall14/handouts/GS-details.pdf