Stable Matching Report

abax, anam, bemn, emba, diem, mkkr & sebni

October 27, 2020

1 Results

Our implementation produces the expected output on all input files.

2 Implementation details

The implementation is written in python. Because of this, we made some modifications to the implementation suggested in Section 2.3 of Kleinberg and Tardos, *Algorithms Design*, Addison–Wesley 2013. Additionally, we used an OOP-approach by subdividing the problem into different classes; `Proposer`, `Receiver`, a collection for each side (Respectively `ProposerCollection` and `ReceiverCollection`), as well as a class for the actual implemention of the Gale-Shapley algorithm (`GaleShapley`). It should be mentioned that the `Proposer` represents the proposing side of things (men) in the example, and the `Receiver` represents the side being proposed to (women) in the example.

We store indices for the proposers and receivers within dictionaries (mapping to objects) instead of the suggested integer mappings. The preferences of the proposing side is stored in a list within the `proposer` object (Which in turn is stored in a dictionary within the `ProposerCollection`) - referring to the receiver objects, not their ids.

Within the `receiver` object we store the ranking for the different proposers using a dictionary to map the index of *proposers* to their individual ranking. Additionally, the *receiver* object stores a reference to the object representing the proposer which they are engaged to, in the field `match` (if they are engaged to any, if not this value is `None`).

It is here important to note that we reverse the preferences of both the proposing and receiving side when loading files. In the case of the input of the *receivers* it's a convenience since we think of *more attractive* as being a higher value (This could however be omitted if the meaning of *likesMore* changes). In the case of the *proposers* it's for convenience when pushing and popping values from their *preferences* stack (This could also be omitted if the push/pop is made on the other side).

At last we also store the available proposers in the `ProposerCollection` in the form of a list (single-linked list as it is the standard python implementation) from which we remove proposers once they are matched with a receiver, or append them when unmatched.

Despite these changes, we can still perform the required operations in constant time:

1. We need to be able to identify a free [proposer]
2. We need, for a [proposer] p, to be able to identify the highest-ranked [receiver] r to whom it has not yet proposed.
3. For a [receiver] r, we need to decide if r is currently engaged, and if this is the case, we need to identify its current partner.
4. For a [receiver] r and two [proposers] p and p', we need to be able to decide, again in constant time, which of p or p' is preferred by r.

(1) is ensured by the list with all available proposers.

(2) is ensured by the list stored in each proposer, from which we (like a stack) can remove (pop) the highest ranked receiver in constant time.

(3) is ensured since we can access any given receiver in constant time through the dictionary stored in `ReceiverCollection`, which in turn has a reference to its current matched `Proposer` object - if it is currently engaged.

(4) is ensured by the rankings stored in each `Receiver` object.

Since we uphold these different criteria, and otherwise follow the implementation of the Gale-Shapley algorithm specified on page 30 (Section 1.1) of Kleinberg and Tardos, *Algorithms Design*, Addison–Wesley 2013. Because of this, the worst case running time of our implementation is $O(n^2)$.

Loading the information, is also $O(n^2)$. Since that's the size of the input supplied. Even given n strings of preferences, splitting them to get the preferences for each requires linear time to do (split is linear on its input size, which in our case is bounded by a linear function of n)