CS 180 Homework 1

Nevin Liang, Nhat Ho, Ty Koslowski

January 9, 2020

Abstract

Solutions to a few problems on Stable-Marriage, Complexity, and Algorithms. Our preference for grading is Problems 3 and 4.

1 Quiescence of Gale-Shapley

Assuming knowledge of the problem, we can identify a few key ideas that help prove that the Gale-Shapley algorithm applied on the Stable-Marriage problem does indeed reach quiescence.

1.1 Monotonically Decreasing Preferences

Let us take a look at things from the man's perspective. Each man has a list of women that he prefers, ranked from most preferred to least. The algorithm states that he will move down this list as he is rejected. Since this is monotonically decreasing, as the man can never go back up his list, there are two following cases. Either the man gets stuck on the same woman forever, or he continually goes down his list until he reaches the bottom. Let us take a look at each of these cases.

1.1.1 In Which the Bottom of the List is Reached

If the bottom of the list is reached, then every woman above on the list has rejected him (and therefore has found someone better, but that is irrelevant). His last message to the woman at the bottom of his list has two consequences. Either a rejection, because the woman has received a proposal from a man that ranks higher on her preferences, or no response (a soft acceptance).

1.1.2 In Which a Man is Stuck Waiting Forever

We have to entertain the idea that a stall can occur, where a man waits for a woman's response indefinitely, as this could prevent the algorithm from reaching quiescence. However, it is not possible for this to happen because each woman ranks men where each man on her list is strictly greater than each man below. If man A were to propose to woman B, after all men have proposed to her, she will choose the one that is the best. It is not possible for her to never respond to a man after all men have proposed.

1.2 Final Solution

Having considered both cases on the process of moving down the list for each individual man, we see that no matter what the woman the man is currently proposed (engaged but not married) to will either stay still until all n men have completed the process, in which case she will be the person he marries, or it will reach the bottom, in which case the algorithm will stop.

2 A Variation on Hal's Theorem

Let us first incorporate a visual to pair with our newly created algorithm for creating a matching (Figure 1).

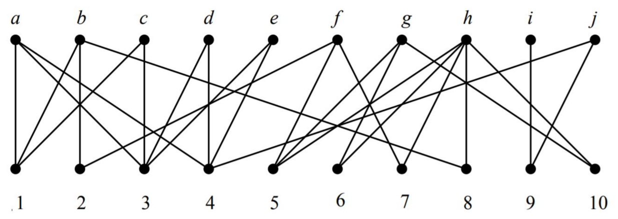

Figure 1: A bipartite graph where letters represent men and numbers, women.

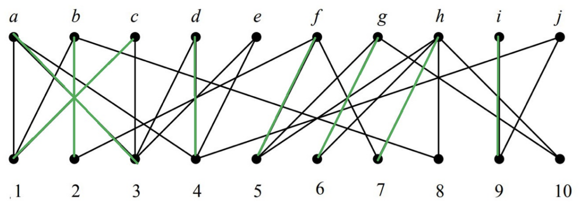

Figure 2: A bipartite graph with matches highlighted.

2.1 Context of a matching

A possible matching (set of marriages), means that we connect one of each letter to one of each number, and this is only possible if we run Hal's Theorem (Algorithm) on the bipartite graph and see that for every subset of men, the neighborhood of that subset of men in the women-space is larger than or equal in magnitude. Figure 2 shows a possible matching, and we can verify that Hal's Theorem is indeed true.

2.2 Description and Key Points of the Variation

The variation in the problem states that now we have the ability to mark a member of the opposite sex as undesirable. This implies the following scenario: pretend that for a certain bipartite graph, there is only ONE possible matching. Pretend this matching involves man a marrying woman 2. In this case, if 2 were to find a undesirable, then we can safely assume that there is NO possible matching, since the matching before was the ONLY possible matching.

2.3 Solution to the Variation

Note: If person x finds person y undesirable, we can pretend that y finds x undesirable as well because it is not possible for them to match anyways.

Suppose we are given a set of undesirable pairs (alpha, num), (alpha, num), ... All we have to do in order to see if the original graph with this constraint is possible to find a match is to preprocess the graph. Before running Hal's, we take away all edges that the undesirable pairs have. For example, if (a, 3) is part of the set, then we remove the edge between man a and woman 3 if there is one. If there is not we just leave it the way it is. After removing all edges, we then rerun Hal's algorithm to see if the inequality still satisfies for every subset of men.

A key example to note is that when we have a bipartite graph of edges between men and women, pretend that there is only 1 valid matching. Pretend that this matching includes man A and woman B. Pretend A finds B undesirable. Then, there is actually no possible matching between people. In fact, if this ever does happen, where a single undesirable condition causes the entire problem to fall apart. Thus, we need to take away all edges of undesirable conditions and then rerun the algorithm. Only when Hal's Theorem returns true is the condition for a stable matching possible.

3 Reducing the Upper Bound

Currently, we have the problem of adding the number 1 n times to a random number. In lecture, we figured out that the complexity of this algorithm was bounded from above by $O(n\log(n))$.

3.1 Explanation of Old Complexity

The complexity of incrementally adding a 1 to a number has a complexity of $O(n\log(n))$ because the time it takes to loop through all the bits of an integer n is $\log(n)$. We are doing this n times, so the total complexity is $n\log(n)$.

3.2 A Possible Lower Upper Bound

We take advantage of the key point that if we add the number 1 to an even integer, because the even integer has last bit of 0, it will just become a 1 and we do not have to loop through all the bits of the integer. Thus, every other number has complexity $O(1)$ and the odd numbers, when added by 1 have a complexity of $O(\log(n)$. When we increment n times, exactly $n/2$ of the iterations are adding 1 to an even number, and $n/2$ are adding 1 to an odd number.

Thus, we have a new, lower upper bound of

$$O\left(\frac{n}{2}\log(n)+\frac{n}{2}\right)$$

4 Discovering the Binary Representation

4.1 Recursive Approach

The recursive algorithm to discover the binary representation of a number is to continually divide by two, and when the current number is odd, place a 1 in the binary representation. Place a 0 if the current number is even.

1. Check if the number is odd or even.
2. If odd, then place a 1 at the end of the binary string.
3. If even, place a 0 at the end of the binary string.

Algorithm 1: Recursive Binary-representation Creation

```
function: GETBINOF(x) := string
if $x = 0$ or $x = 1$ then
    return $x$
end if
if $x$ is odd then
    return GETBINOF($\frac{x-1}{2}$) + '1'
else
    return GETBINOF($\frac{x}{2}$) + '0'
end if
```

This algorithm determines the binary representation by incrementally adding on either 1 or 0 to the end of a string depending on the value of whether it is odd or even. At the end of the recursion dive, when it comes back up to the surface, it will add 1 or 0 to the previous function call's value, and when it returns from the top-most function, it will return the string that represents the binary representation of the integer. Since it divides by 2 every time it calls a child function, the complexity of the code should be $O(\log(n))$ where log is base 2.

4.2 Iterative Approach

An iterative approach would just be to use a while loop instead of recursion. This would basically be the following:

Algorithm 2: Iterative Binary-representation Creation

```
function: GETBINOF(x) := string
answer = ""
while $x > 1$ do
  if $x$ is odd then
    append 1 to answer
  else
    append 0 to answer
  end if
  $x = x/2$ (integer division)
end while
append $x$ to answer
reverse answer
```

This algorithm does basically the same thing as the recursive one except it technically runs backwards, and appends in the wrong order, so at the very end you have to reverse the string to get the correct binary representation. The complexity of this iterative method is the same as that of the recursive approach, as it is still dividing by two every time, however the recursive approach has more overhead as well as does the path-ing exactly twice: one down and one back up. The iterative approach only does this once, so the theta is better.