Optimizing and Analyzing the Worst-Case Runtime of Algorithms Case Study: The Gale-Shapley Algorithm

Shaddin Dughmi

August 24, 2020

The purpose of this note is two-fold:

- Illustrate the importance of the careful choice of datastructures in the design of fast algorithms.
- Explore some of the nuances of runtime analysis.

We will use the Gale-Shapley algorithm from class, restated below, as a case study for both goals.

Input: A set M of n men and a set W of n women.

Input: For each man $m \in M$, a total order $\succeq_m$ on W, and similarly for each woman w a total order $\succeq_w$ on M.

1: Let $S = \emptyset$ be the set of engaged couples
2: **while** $\exists m \in M$ who is free (i.e. not engaged in S) and hasn't proposed to all the women **do**
3: Select such a man m.
4: Let w be the highest ranked woman in $\succeq_m$ to whom m has not yet proposed. Issue a proposal from m to w.
5: **if** w is free **then**
6: Designate m and w as engaged (i.e., add (m, w) to S).
7: **else if** w is engaged to $m' \succeq_w m$ **then**
8: Do nothing
9: **else if** w is engaged to $m' \preceq_w m$ **then**
10: Break the engagement of m' and w (i.e., remove (m', w) from S)
11: m and w become engaged (i.e., add (m, w) to S)
12: **end if**
13: **end while**

Output: The set of couples S

We proved in class that the Gale-Shapley algorithm terminates after $O(n^2)$ iterations of the while loop in the worst case. By itself, however, this does not guarantee a worst-case runtime of $O(n^2)$, since we have not established any runtime bounds on each iteration. For example, one naive implementation of the algorithm would select m and w (steps 3 and 4) using two nested searches, with an outer loop traversing the list of men looking for one who is free, and an inner loop traversing

the free man's preference list searching for the top woman to whom he has not yet proposed. Such a naive implementation would take time $\Omega(n^2)$ per iteration of the while loop, for a total runtime of $\Omega(n^4)$, in the worst case.

It is hopefully now clear that implementation details, which we sometimes omit in high level pseudocode, are nevertheless often crucial to obtaining a fast algorithm. In the case of the Gale-Shapley algorithm, we can implement each iteration of the while loop in $O(1)$ time in the worst case by maintaining some simple but carefully chosen datastructures and pointers. In more detail, we need to implement the following operations in $O(1)$ time:

1. Find an arbitrary free man if there are any, and add or remove a given free man from the pool of free men.
2. Given a free man m, find the highest ranked woman to whom he has not yet proposed.
3. Determine if a given woman w is free, and if not find her partner.
4. Compare two men m and m' in a woman w's total order $\succeq_w$.
5. Add a couple to S, or remove a given woman w's couple from S.

For (1), we can maintain a doubly-linked list of the free men, which permits all three operations in constant time. For (2), we can maintain each man's total order $\succeq_m$ as a linked list, and maintain a pointer to the position in that list of the next woman m should propose to, advancing that pointer with each proposal. For (3), each woman can maintain a pointer to her partner, or to null if she is free. For (4), we can maintain a two-dimensional array $rank$, with $rank[i][j] = k$ if man m_j is woman w_i's kth favorite man. This permits comparing m' and m by comparing the two integers $rank[w][m]$ and $rank[w][m']$. The array $rank$ can be initialized in time $O(n^2)$ at the very beginning of the algorithm. For (5), we can maintain S as a doubly linked list of couples, and maintain a pointer from each woman to her couple in S (or to null if w is free).

Given the above datastructures, we obtain $O(1)$ runtime per iteration, for a total runtime of $O(n^2)$ for the Gale-Shapley Algorithm. Or do we??

As we often do, we have swept some details under the rug here. In particular, we implicitly assumed that the following operations take constant time:

- Read, write, and follow a pointer to a position in memory (e.g. a woman's pointer to her current partner)
- Basic arithmetic operations on integers (e.g. comparing the ranks of m and m' in a woman's preference list)

Are these assumptions realistic? If one were to be really strict and pedantic, the answer would be no: It takes $\Omega(\log N)$ time to add, multiply, or compare integers between $-N$ and N. Moreover, it takes some time for a disk head to move to a position on the disk, linear in the amount of continguous working space being used by the algorithm. Even reading a pointer to a position in memory would take time logarithmic in the total size of memory being used!

However, you have been taught in previous classes to assume that these operations take constant time when tallying an algorithm's runtime, and we will continue this tradition in this class. This is almost always justified in practice: modern machines are typically equipped with random access memory, and the number of bits used to express integers is small enough (e.g. 64 bits) so as to make basic arithmetic operations essentially instantaneous. Even in purely theoretical research, it is typical to assume that such basic operations take constant time. This is formalized by what we

refer to as a "machine model", the most famous example of which is the Turing Machine which you will see at the end of this semester. Loosely speaking, a machine model is the interface between the world of algorithms and the physical world, and essentially encapsulates a set of assumptions on which operations we call "basic" or "primitive", and how long they take. Analyzing runtime can then be viewed as "relative" to the time taken by these basic operations in the physical world.

We won't bore you with the details of machine models at this stage in your education; just know that they exist as the theoretical foundation of algorithm design and analysis. Also know that we will assume that basic operations, such as reading/following pointers and basic arithmetic, count as constant time. This is "true enough" in practice, and is either true or only logarithmic factors away from being true in suitable machine models, so as to justify our assuming it. There are exceptions, and possible abuses of these assumptions, which should never arise in this class, but are worth being aware of: If your integers get really big, say exponential in the size of your input or bigger, basic arithmetic ceases to be logarithmic time and hence shouldn't be ignored when analyzing runtime. Another exception is when you are designing an algorithm for a basic operation itself, such as integer multiplication (which, actually, was the subject of a recent breakthrough result). In such settings, logarithmic factors in the runtime tend to be of great theoretical interest, and the details of the machine model do become important, and can greatly influence the results of your analysis. All that being said, for most algorithmic tasks you will encounter as a computer scientist, whether in the real world or even in research-level CS, assuming that the usual basic operations take constant time is a well-accepted convention which we will follow in this class.