On the Science Behind Computing (Part I)

Samir Khuller *

1 Background: Why a New Course?

The basic idea behind the new course is to present the science behind computing. First of all, this is already a daunting challenge – to try and condense decades of research on computing into one small semester long course. There are too many aspects of computing ranging from the design of the actual hardware (building computers), to design of systems software (software that manages your computer), to dsigning applications, not to mention the Internet and a host of other issues such as defining computation, understanding the limits of computation etc. The attempt here will be to introduce specific nuggets of information that will convey a sense for what the field of computing is about and this will hopefully provide a glimpse into the mind of a computer scientist and our way of looking at the world.

Many people use several software packages, without really having an understanding as to how these work, or without a complete understanding of *what* exactly happened, let alone *how* it happened. There is no sound of the engine revving up, or the squeal of the brakes, or the smell of burning rubber as in a powerful racecar. Thus we cannot "feel" computation. Since the underlying software is not something people can "see", there is little appreciation for what is involved in building software tools that work efficiently and correctly, unlike appreciating an impressive building's architecture, or looking under the hood of a sports car! For example people use online maps, get directions, use word processing software, search for files on their computer containing keywords, use search engines etc. They simply click a button and the answer comes back promptly without giving a clue as to what really happened in that fraction of a second. Computing has become like "electricity", you flick a switch and something happens. Except that now you don't just get one thing, but access to a world of information.

Moreover, there is even less understanding about the underlying *algorithms* that drive software packages. All software packages that come on a CD drive, look identical. Why is it that it takes years of effort, with hundreds of programmers all working day and night to produce software that goes into a single CD? What are all the thousands of engineers at Google, Microsoft and IBM doing? They are busy solving problems, designing systems and efficient methods that takes years of effort to engineer and build.

One of the goals of this course is to essentially show how a lot of the tools we use, those that have become part of our everyday life, really work. More surprisingly, how they can be mapped to rather easy to state mathematical objects and problems. These mathematical problems appear to be toy like sometimes, but the algorithms that have been developed to solve them, are incredibly powerful, and can be used to build rather complex and useful tools.

*Department of Computer Science, University of Maryland, College Park, MD 20742. E-mail : samir@cs.umd.edu.

2 Introduction to Algorithms

Algorithms can be thought of as a set of rules that specify how to compute the desired output, from an input to a problem. A simple algorithm might describe how to bake a cake using a set of ingredients. The set of rules that specify how to convert a specified input into the desired output is the algorithm itself. Algorithms can be described at a variety of levels. Our descriptions, will mostly be given in English. A computer on the other hand needs very careful and precise specifications about the algorithm. Even a small mistake, or imprecise specification can create unexpected and strange behavior from a computer.

It is important here to note the distinction between an *algorithm* and a *program*. An algorithm is a set of rules that tells us *how to perform the computation*, as well as *what computation to perform*. A program on the other hand, is the embodiment of this set of rules in a specific programming language that can be executed on a computer. It is similar to a story, that can be written and translated in several languages. The story corresponds to the algorithm, and the description of the story in a specific language corresponds to a program that implements the algorithm. At the lowest level[1], a programmer could describe her algorithm in *assembly* language, a very simple language with very simple instructions. Programs written in this language are somewhat long, and tedious and hard to understand. Thus most people use high level languages that make it easy to specify the algorithm. Compilers are then used to translate the descriptions given in high level languages to assembly language.

The field of algorithms is a little like mathematics. Often the problems are easy to describe, the solutions are sometimes very ingenious, and sometimes even easy to verify by a reader who knows little about the problem! Often it is easy to understand new problems, as well as understand their solutions. However, coming up with the solutions can be quite challenging; and being an expert sometimes is of no help at all. The most creative solutions could come from smart graduate students, who have just started research in the area (often they are the ones with wild ideas, and have the energy to show that some of these wild ideas actually work). In the process of explaining the problems and their solutions, we will see that the reader can learn some useful tools which can then be used to attack other problems.

I do not want to assume a high level of mathematical maturity, so the "proofs" will be informal, usually by simple explanation or examples. The reader is urged to read the technical papers or books referenced at the end of each section to identify further readings.

3 Algorithms Background

Algorithms are simply methods or recipes for solving problems. An algorithm is a step by step procedure that takes an input and produces the desired output. Take for example the simple problem of "given an integer n, is it *prime*?". Recall that a prime number is one that is only divisible by 1 or itself, and no other number. For example, 2 and 3 are prime numbers, but 4 is not prime as 2 divides 4. Again, 5 and 7 are prime, but 6, 8 and 9 are not since 2 divides 6 and 8, and 3 divides 9. For small numbers, it is easy to obtain a listing of all prime numbers by checking for divisors that are smaller than the number itself. However, given a number such as 1019, how would we check to see if it is prime? (Prime numbers have a variety of uses...) One trivial algorithm would be to check if some number i divides it, but we will have to perform this test for each value of $i < 1019$. Clearly 2 does not divide it, and neither does 3. If 2 does not

[1] Strictly speaking this is not the lowest level, since a programmer could also write her program in machine code, a language directly understood by the computer.

divide it, then clearly 4 does not either. In fact, we can generalize this observation to only check for divisibility with known primes. The numbers 5 and 7 do not divide it either. What about 11? Is there no better way? What would we do if we needed to check if a much larger number was prime? While there are fast algorithms for solving this problem, they are out of scope for this monograph.

Suppose we have a street map of Washington DC. We need to get to the airport quickly, in order to not miss our flight. How do we choose a route? People have different ways for doing this. Some rely on local knowledge of the area, and knowledge of traffic patterns etc. Clearly if we have been living in an area for a long time, we can do a good job doing route planning. Suppose we are just visiting a city and renting a car and we have no idea about the traffic patterns? Suppose we just want to simply take the shortest route? Or a route that will get us there in the least amount of time. Often we look at a map and do "visual navigation" to compute a route. One algorithm would be "go from your starting point via some short route to the nearest highway, and then use the highway to approximately where we wish to go, and then get off at the appropriately marked exit, and then look up the map for a short route to the destination". Have you ever thought about how a computer would solve this problem? There are services such as mapquest[2] that gives directions and routes between chosen locations. Of course there is an issue of the "representation" of the map in the computer. One can scan a map in, and this is fine for simply doing a visual display of the map, or for zooming and viewing parts of the map. How about actually doing route planning? For this a different representation is required so that we can manipulate the information in an easier manner, perform computations and solve interesting problems.

How do we know if the algorithm is good or bad? Typically most algorithms are evaluated by the number of "elementary steps" that they take to solve a given problem (also referred to as running time of the algorithm). However, what an elementary step is can be the subject of a lot of discussion. Informally, we will assume that this is an "instruction" that can be carried out in some fixed amount of time. Often, the actual number of steps taken by the algorithm depend on several factors. The size of the input to the algorithm is itself one of the biggest factors determining the running time of the algorithm. Moreover, the running time of the algorithm may vary when given two different inputs of the same size. We usually parameterize the size of the input to the algorithm by a parameter n, and measure the *maximum* running time of the algorithm for any input of size n. The issue is that for different inputs of the same size, the algorithm may have different running times. How do we measure the running time in this case as a function of n? In fact, there may be an infinite number of possible inputs of size n. One option is to try and derive some bound on the *average* running time for all possible inputs of size n. However, this is non-trivial and has only been done for a relatively small number of problems. Another traditional method is to derive an upper bound on the running time over all possible inputs of size n. This is referred to as *worst* case analysis of the algorithm, and is probably the most common method used for analyzing algorithms. However sometimes this can be too pessimistic. There are some classical algorithms, such as the famous simplex algorithm for solving linear programs, which have a very poor worst case bound, but on average they do extremely well and are the algorithms that should be used in practice.

Let us focus briefly on the need to design efficient algorithms. Consider for example the problem faced by a police sheriff in a small town. The sheriff would like to visit every street

[2]http://www.mapquest.com

intersection to check on the traffic lights after a big storm. How does the sheriff plan a route that minimizes the travel time? One approach is to enumerate all possible ways of visiting the intersections, computing the cost of each one. However, this can be a very large number. In other words, even for reasonably small cities, the total number of possible routes is huge and cannot be enumerated efficiently by even a computer. For example, if we are looking to visit a set of n locations, and consider all possible orders of visiting the locations, the number of orders to be examined is $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1$. Even for $n = 10$ this is as large as 3628800. This problem is also popularly referred to as the *Traveling Salesman Problem* and several heuristics are known for it. We will discuss this problem in more detail later.

4 Choice of Problems

There are many sub-areas in the field of algorithms - this can be seen by the proliferation of conferences and workshops in this field such as online algorithms, approximation algorithms, dynamic algorithms, efficient algorithms, geometric algorithms, graph algorithms, string matching algorithms, algebraic algorithms, biological algorithms etc.

To illustrate the beauty of algorithms I have chosen topics that are more readily understandable to a non-specialist, rather than choosing problems that are more important. Sometimes, the background needed to explain a particular result requires more mathematical knowledge than I wish to assume, or simply requires knowledge about other results. I have tried to pick examples that are related to real-life problems, or fundamental problems that are useful in developing methods that are really useful in practice, and at the same time easy to explain. While these are not precisely the problems I work on myself, but they do represent a good sample of the thought processes involved in the design of algorithms.

Algorithms and Computing were initially confined to performing mathematical calculations. A quote widely attributed to Thomas Watson Jr. (president of IBM), is "I think there is a world market for maybe five computers". According to Wikipedia, there is no proof that he ever said this. However, it is based on the fact that computers were rather expensive devices, that were in use for performing primarily mathematical calculations. In time, however, it became clear that computers could process information of any type and not just perform mathematical calculations.

5 Stable Marriage Problem

This problem finds applications in a variety of situations where we are trying to find an "optimal" pairing of people. One situation arises in the assignment of medical interns to hospitals, where an algorithm based on this approach is used, once the interns and the hospitals submit their "preference" lists. The situation is not identical to the one described below; however, the basic approach that is used is very similar to the one described below.

In the most basic version of the problem we are given a set of n boys and n girls. Each girl provides a preference list of all the boys such that the first element in the list is the boy that she likes the most and the last element in the list is the least preferred boy. She is required to rank order all the boys. In a similar manner, each boy rank orders all the girls. The preference lists of the girls and the boys contain all the individuals of the opposite sex. A pairing of each boy with a distinct girl is called a marriage, and the matched pairs are called couples. So for example, let us assume that there are three boys and three girls. Suppose that the boys are called John, Rajeev and Pierre and the girls are called Janet, Saira and Marie. Each boy submits a rank ordered preference list of the girls - so for example, the list submitted by the boys could be:

John: Saira, Marie, Janet.
Rajeev: Marie, Saira, Janet.
Pierre: Marie, Janet, Saira.

The lists submitted by the girls could be:

Janet: Rajeev, Pierre, John.
Saira: Pierre, Rajeev, John.
Marie: John, Pierre, Rajeev.

A marriage is a pairing of each boy with a distinct girl. A marriage is called *unstable* if the following is true: There is a pair of individuals, who are not married to each other, but both prefer each other to their current mates. So for example, the marriage $(John, Saira), (Rajeev, Janet), (Pierre, Marie)$ is unstable since Rajeev clearly prefers Saira to his current partner Janet. At the same time, Saira prefers Rajeev to her current mate John. This is an unstable marriage since both have an incentive to leave their partners for each other. The marriage is called unstable, regardless of how the jilted partners John and Janet like each other.

The most interesting fact is that a stable marriage always exists! Moreover, there is a simple algorithm that can find it. (If only Computer Scientists were allowed to run society, and people's preference lists did not change!)

The algorithm proceeds in parallel steps, with the boys making proposals. Even though we describe the method as a synchronized process, actually it does not really matter for the algorithm in what order the boys make proposals. Initally everyone is unengaged.

Algorithm for finding a stable marriage

Algorithm 5.1.

Each currently unengaged boy proposes to the most preferred girl that he has not proposed to until now.

Every girl who receives a proposal checks her preference list. If she is unengaged, she accepts the best proposal, turning down the other proposals that she receives. If she is currently engaged, she breaks the engagement if she receives a better proposal and accepts the new proposal, and gets engaged to the most preferred boy among the ones who proposed to her.

In the next round, the boys who lost their fiancés as well as the ones who are unengaged, will propose. If ever we reach a situation where everyone is engaged then the algorithm halts and outputs this solution as a stable marriage.

We will illustrate this algorithm by an running it on the above example to make it clear. In the first round, all three boys propose to their top choice. Thus John proposes to Saira, Rajeev to Marie and Pierre to Marie. Note that Saira receives one proposal and accepts it, getting engaged to John. At the same time Marie receives two proposals, and prefers Pierre to Rajeev. She gets engaged to Pierre and turns down Rajeev's proposal. In the next round, Rajeev proposes to the next girl on his list, namely Saira. Saira breaks her engagement with John, since she receives a preferred proposal and gets engaged to Rajeev. In the next round, John proposes to Marie, the next girl on his list. Marie prefers the new proposal from John, and breaks her engagement with Pierre and gets engaged to John. In the next round, Pierre proposes to Janet. This is the first proposal that Janet receives and she accepts it. All parties are now engaged and we

	Round 1	Round 2	Round 3	Round 4
John	Saira √	~~Saira~~	Marie √ →	Marie
Rajeev	Marie ×	Saira √ →	Saira →	Saira
Pierre	Marie √ →	Marie →	~~Marie~~	Janet √

KEY
× = rejected immediately
√ = accepted
~~ ~~ = rejected (after acceptance)
→ = relationship continued to next round

Figure 1: Running the stable marriage algorithm.

perform the wedding ceremonies. We end up with the marriage: (Pierre,Janet), (Rajeev, Saira) and (John,Marie).

The above algorithm will always terminate: because, when a boy proposes to the last girl in his list all the other girls are engaged (to the other $n-1$ boys), so the remaining girl (who is not engaged) has to accept his proposal.

PROPOSITION 5.1. The marriage found by the above algorithm is *stable*.

Proof. We can informally argue this as follows. Suppose the algorithm ends up with a marriage. Assume that there is a boy A who prefers Y to his partner B. At the same time Y prefers A to her partner X. Since A prefers Y, A must have proposed to Y before proposing to B and was turned down (either immediately, or was engaged to her for a while before Y broke the engagement). In either case, Y gets a partner who is preferable to A. Hence the partner that Y ends up with, must be a preferred choice over A. This is a contradiction to the assumption that Y prefers A to her partner X.

To see this point, observe that in the stable marriage produced by the algorithm there is no unstable pair. For example, note that Rajeev prefers Marie to Saira, his wife. However, he did

propose to Marie, who turned him down for Pierre, and ended up with an even better partner John (from her point of view). In fact, when a boy ends up with a girl who is not his first choice (for example John ends up with Marie), there is no stable marriage at all with someone he prefers to the partner he gets. In particular, there is no stable marriage at all in which John is paired with his first choice Saira.

Interestingly, the algorithm can also be run with the girls proposing to the boys. This may produce a *different* stable marriage. In fact, the solution is far from unique. There are situations when there is a large number of possible stable marriages. In fact, the sex that proposes has a distinct advantage in this approach. With the way in which we have described the algorithm, the boys end up with the best choices they possibly can in any stable solution. The girls end up with the worst possible solution from their point of view! In fact in the example given above, note that if the girls are the ones proposing, after the first round itself the algorithm terminates and each girl gets her first choice.

6 Bipartite Matchings

Let us consider another type of assignment problem, in a sense similar to the stable marriage problem. Suppose we have a set of workers W and a set of jobs J. For example $W = \{Bill, Jill, Tom\}$ and $J = \{Mulching, Planting, Trimming\}$. Moreover every worker is not qualified to perform each task. For example Bill can only do mulching. Jill can do all three jobs, and Tom can either do planting or trimming. How can we assign each worker to one job that they are capable of performing? A matching is simply an assignment of workers to jobs, so that each worker is assigned to at most one job, and each job is assigned to at most one worker. A perfect matching is a matching in which every worker is assigned to a unique (and distinct) job, and each job has a worker assigned to it. For convenience we will assume that the number of jobs and workers is the same. There are situations when there is no perfect matching. For example if both Bill and Jill can only do mulching, then there is no perfect matching. We can model this situation as a graph. There is a set of nodes W, one node for each worker, and a set of nodes J, one node for each job. We add a link between a worker and a job if the worker can perform that job. Given an arbitrary graph structure of this type we would like to quickly compute a perfect matching (note that the perfect matching is not unique, and several different solutions may exist). For example, we can have a solution $\{(Bill, Mulching), (Jill, Planting), (Tom, Trimming)\}$ or $\{(Bill, Mulching), (Jill, Trimming), (Tom, Planting)\}$. While it is easy to find all the solutions for such a small graph, how do we solve the problem quickly when we have hundreds of workers and jobs? Or even thousands of workers and jobs?

Of course one can try all possible ways of assigning workers to jobs, and then check which ones are feasible assignments. Such an algorithm while easy to implement on a computer, can actually be extremely slow. The running time of such an algorithm is $N!$, where N is the number of jobs (or workers). There are many situations where we need to find a maximum (or perfect) matching in a graph, so it will be incredibly useful to develop such an algorithm. Below we describe precisely such an algorithm.

We will illustrate this algorithm by showing how to apply it to the “prom problem”.

Lets suppose there are four girls and four boys. Each person specifies a subset of people they are willing to go to the prom with.

For example Margaret, Jane, Amy and Katherine would all like to go to the prom with someone they like. Suppose Margaret is willing to go with Tom or Bill. Jane will only consider going with Bill. Amy on the other hand is willing to go with either Bill, Tom, or Don. Finally,

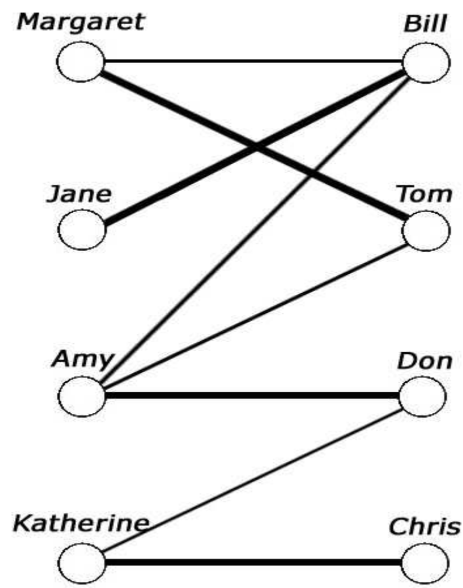

Figure 2: The graph construction.

Katherine is willing to go either with Don, or Chris. Each girl lets their preferences be known to their mothers. The mothers all meet at Starbucks one morning and are trying to figure out if there is a solution in which each girl can ask someone to go to the prom with her, so that their offer is accepted. Lets assume that the boys will accept any proposal that is made, if they do not get asked by another girl within a day of getting asked by some girl. The mothers on the other hand, do not want their daughters to have their offer rejected as it could lead to a tense situation!

The mothers come up with the following matching (Margaret, Tom), (Jane, Bill), (Amy, Don), (Katherine, Chris). Each mother tells their daughter to make the appropriate proposal, and is pleased to find out that her daughters proposal was accepted a day later.

How did they come up with this solution? To describe the algorithm, lets play the following mental game: supppose Margaret asked Bill and he accepted. Amy asked Tom and he accepted. Katherine asks Don and he accepts. Finally, Jane asks Bill. (Even though no-one has asked Chris as yet, Jane does not want to go with him and does not ask him.) Bill says that he will accept Jane's offer, provided he can get a friend to go with Margaret (and it has to be someone Margaret likes). Bill then asks Tom to go with Margaret. Tom has already said yes to Amy, but again is willing to go if he can find someone who Amy likes. Amy likes Don, but Don has already said yes to Katherine. Now we can "shift" Katherine to Chris, freeing Don. Amy can now go with Don, freeing Tom, Margaret can go with Tom, and Jane finally gets to go with Bill. However, if the mothers knew all this before, they could confer and compute this solution before any actual proposals are made. Each person then only makes one offer, which is accepted. The main idea is that we can start with any initial pairing, that matches k compatible pairs together. The algorithm can then iteratively "improve" this pairing to add one more valid pair. However, this involves changing the current solution. This process continues until no further improvement is possible. At that stage we can prove that if the algorithm fails to increase the number of compatible pairs, then no better solution exists.

Suppose we change the preference lists in the following way: suppose Amy hears that Don is not such a nice person after all, and does not want to go with him. The current solution might be (Margaret,Bill) (Amy,Tom) and (Katherine,Chris). Jane can only go with Bill and if we shift Margaret to her remaining choice Tom, then there is no other valid choice for Amy. In this case, the algorithm fails to find an improvement, because none exists. In fact, note here that the three girls, Margaret, Jane and Amy, are only willing to go with eithe Tom or Bill. Clearly there can be no solution which finds a valid pairing for all three girls since they are interested in only two boys. In fact, Hall proved that there is always a perfect matching unless there is a collection of p girls, such that there are less than p boys they are willing to go with. (In this situation, $p = 3$.)