CS 51 Final Project Report

Luis Perez | luisperez@college.harvard.edu
Andy Shi | andyshi@college.harvard.edu
Zihao Wang | zihaowang01@college.harvard.edu
Ding Zhou | dzhou@college.harvard.edu

May 5, 2013

Link to demo video:

1 Overview

In the real world, people love to make money. Even in virtual worlds (bitcoin?), people love to make money. Basically, many problems in business and even everyday life (but especially business), boil down to maximizing profit or minimizing costs. Of course, when you speak of minimization or maximization there are always constraints. Resources are finite and time is precious. Therefore, in an abstract sense, the problem boils down to minimizing or maximizing some quantity under certain constraints. For example, you might want to maximize the number of textbooks x, y, z you buy under the constraints that you can't buy more than five from one company, the total cost can't be more than \$500, and the last two books you buy must cost more than \$75. This problem is precisely the type of problem that our program can now solve. How did we do this?

To start off, we created a framework. We implemented a matrix module with standard matrix operations such as multiplication, addition, map (where scaling follows) as well as many extra features such as row reduction, determinant, trace, and many more. The matrix module basically allows a user to do almost anything imaginable with a matrix. In addition, we provided read and write functionality for easy loading of data, and printing functionality for straightforward viewing of the abstract matrix. However the intellectually challenging part of our project is the implementation of the simplex algorithm. We needed to create a matrix module simply because OCaml doesn't provide one (also because we just wanted to implement a matrix module, of course). We use many of the functions in the matrix module to create and manipulate simplex systems within our simplex module. This allowed us to abstract away yet another part of our project.

The simplex algorithm is a type of linear programming that allows one to optimize a linear equation given linear constraints (like the example problem at the beginning). In mathematical terms, the simplex program operates on linear programs in standard form. That is, it seeks to minimize (or maximize) $C\mathbf{x}$ subject to $A\mathbf{x} = \mathbf{b}$ (Wikipedia). $\mathbf{x}$ and $\mathbf{b}$ are vectors. x contains the variables in the equation we want to optimize (the objective), and b contains the constants in the constraints. C and A are matrices, and they represent the coefficients in the objective and the constraints, respectively. Let's consider the inequalities: $30x + 15y + 18z \geq 500, x \geq 5, y \geq 5, z \geq 5$, and $15y + 18z \geq 75$ and the objective functions $x+y+z$ where $x, y, z \geq 0$. As you will notice, this is precisely the system of equations necessary to solve our book example above. The algorithm converts the inequalities into equalities by adding positive slack variables. Then it represents this set of equations in the matrix form where the coefficients of the terms we maximize are on the first row and each column represents a variable. This provides affordable storage of most of the information necessary to solve the linear program. Furthermore, we then define basic and non-basic variables (kept track of in a list), which together with the aforementioned matrix, create a simplex system.

The first function necessary for simplex is called `simple_solve`. This function, in abstract terms, finds the minimum value of a passed in system (as described above). When we call the function `simple_solve`, the program finds the entering and leaving variables which are the variable that would exchange position from basic and non-basic (slack to non-slack in the initial case). Here, the algorithm is making the assumption that the system has a feasible solution where all of our non-basic variables are zero, and therefore exchanges the role of one of these variables with the corresponding basic variable. In geometric terms, we are moving along one edge of the n-gon formed by the constraints and attempting to minimize. This step is accomplished by the helper function pivot. The function `simple_solve` continues to loop (moving along edges and minimizing) until the first row—which represents the term we want to optimize consists of only non-positive values or there is no possible entering variable (ie, any change in the variables will only increase our answer, not minimize it). In the first case, algebraically, it means that the constraint can only increase if the variables change. Therefore, we have found our minimum value which is the constant stored at the end of the first row. This is the optimized value of the objective function. In the second case, where no suitable entering variable can be found, it means that the system can be decreased infinitely as pleased and therefore it is Unbounded. In the case where a solution exists, one possible value for the original variables can also be found by looking at their corresponding columns and checking to see if they are basic. If there are an infinite number of these, our algorithm returns only one.

What if we had the constraint $x + y + z \geq -10$? As stated above, our `simple_solve` makes the assumption that one of the feasible solutions is the zero solution, but if we initially set x, y, and z to 0 (our algorithm generates a "basic solution" by doing this step) the constraint does not hold. We therefore have a helper function called `initialize_simplex` that converts everything into the "simple" form so we can just recursively call `simple_solve`. It does this by following a very complex algorithm better detailed below. Our final simplex algorithm now accounts for all cases as long as the equations are linear. It also handles not only less than or equal constraints, but also greater than or equal to and equality constraints. Finally, not only can it minimize, but it can also maximize. You can follow the algorithm at each step on www.simplexme.com (we used this to debug parts of our code), but we found that SimplexMe only works for the "simple" case. It has errors for large matrices and the infeasible and unbounded cases.

2 Planning

Here is our functional spec and our technical spec (the links open new PDFs).

We planned to have a completed Simplex solver for simple cases, handling less than or equal constraints in addition to a complete Matrix module. Here are our initial functional spec and technical spec. We achieved far more than we expected to achieve.

Our matrix module is fairly robust and it certainly implements everything that is necessary to program the simplex algorithm (which was our main goal for this module). Yet, it goes further. We managed to implement determinant, trace, row-reduction,inverse, LU decomposition, and more (all extra functions that we classified under our "cool features" category. There were only minor points that we did not achieve, such as the eigen function, which finds eigenvectors and eigenvalues.

As to the Elts module, we really went far and beyond. Because we coded everything from an abstract point of view, once we had completed our project we were able to go back and add a new Nums module which provided arbitrary precision and size (ie, implementing arbitrary float arithmetic and bignum arithmetic). Of course, to be honest, we really didn't actually code most of this up. We used the built in Nums library. But as can be seen inside of Elts.ml, our code was planned out so well that we can replace the foundation of the mathematics in our program (`ORDERED_AND_OPERATIONAL`) with only a few changes in code.

The Simplex module also went further than expected (though it certainly was more difficult to implement than we expected). Not only did we implement the ability to solve simple linear programs, but we can now solve with certainty any linear program (dual and primal—dual requires `initialize_simplex` while primal is the "simple" case), and provide the user with not only the optimal value, but also one possible solution (which, in most cases is the only solution).

Most of the milestones were achieved for all of our individual modules, with the exception of our `row_reduce` (which was buggy and took longer than expected), and of our simplex module (we went a day over what we had originally planned).

3 Design and Implementation

Implementing simplex was difficult. A lot of difficulty stemmed from the complexity and size of the algorithm itself, and we spent a lot of time trying to understand each step. We never completely internalized every step of the algorithm which made debugging difficult because it took a lot of effort to recall the correct steps. We also had some initial terminology confused which caused some of the code to be written incorrectly. Since each step ended up being so complex, whenever we moved to the following steps of the algorithm we always had trouble recalling what exactly we were doing and if someone else had already done it. Furthermore, bugs were very difficult to find because the system was so complex and there weren't many cases for which we could work out the steps by hand.

Finding SimplexMe was essential to finally getting all of the bugs out of our system, but it also added many hours of unnecessary debugging. SimplexMe has implemented simplex incorrectly and therefore the answer it gives for non-simple systems can be, at times, incorrect. We verified this with Mathematica's built in Linear Programming function.

Testing the system also proved to be a problem in its own right. Even after we got all of the parsing functions down and were certain that the data was being loaded correctly, we were never really able to find a large set of simplex example problems. In the end, we resorted to each team member individually finding a problem and solution (with Mathematica, of course, since SimplexMe proved unreliable), and then typing out the data into a text file so we could test it. Once we had all the data, we decided to program an interface that would allow for expanding the testability of the code. More information on this can be found in the `README`.

To use the simplex algorithm, we first parse a standardized plain text file containing the parameters of the objective and the constraints. The first line of this file must either be `max` or `min`, signalling a maximization or minimization step. The next line denotes the objective function. The coefficients of each of the variables in the objective are separated by columns (decimals must be represented by fractions because we are using Ocaml's num library). The last element in this is the "constant" (5 if the objective is $2x + 3y + 5$). The next line must read `subject to` to help us with parsing. The next lines are the constraints, where the coefficient of each variable that appears in the objective must appear (in the same order as in the objective), separated by commas. We also allow the user to enter if the constraint is a $\geq, \leq$, or $=$ condition. Again, the last element is the constant (5 in the case of $x + y \leq 5$). It is assumed that all the variables in the objective function are ≥ 0.

Our program will parse this text file and generate a matrix representing the objective and the constraints. Additionally, our program adds slack variables (for example, (we change the inequalities in the constraints to equations—$3x + 2y \leq 5$ becomes $3x + 2y + s = 5$, where $s \geq 0$). This makes the problem easier to solve).

The simplex algorithm basically takes the objective function and attempts to find a "basic solution" which will satisfy all the constraints. Even if this solution is trivial (like setting all the variables in the objective to 0), we take this solution and keep optimizing it by replacing a variable not in the objective function with a variable that is. We check the objective to make sure none of the coefficients are negative; if they are, we are finished and we have the optimum value in the top right corner of the matrix (because we can set all the variables in the objective to 0 and not violate the constraints). Our "simple" case is the case where we want to minimize the objective and all the constraints are $\leq$ relations. However, we can change any problem to this form. Maximization can be achieved by minimizing the negative of the objective. Similarly, a $\geq$ constraint can be changed to a $\leq$ constraint by multiplying each side of the inequality by -1. Finally, an $=$ constraint can be changed into a $\geq$ and a $\leq$.

As to the design and implementation of our actual code base, it definitely can be improved. It is the best it can be given the fact that we hadn't already done this same problem before, but as with anything, it can be better. For example, our `initialize_simplex` function ended up being very complex and complicated. We all understand the general idea of what it does, but not all of us could implement it by ourselves from scratch. Basically, `initialize_simplex` takes in a system and adds the slack variables to it. During this step, if the algorithm determines a system (the objective function and the constraints) to be unfeasible, it adds an extra variable and attempts to minimize that (ie, replaces the objective function). If this new system has a minimum of 0, then the original system is feasible and all we need to do is substitute the old objective back in appropriately. Otherwise, the original system is unfeasible and the program returns `None`. In the first case, an extra pivot operation might be necessary, but again, `initialize_simplex` takes care of this. In the end, it calls `simple_solve` (the function we originally wrote) on this now simple case and a solution is produced. However, we did run our simplex algorithm against an equation with 100 variables and 100 constraints (`test22.txt`), and it solved it in under 1 second. Even though the worst case running time of simplex is not polynomial (it's exponential), on most real world cases simplex runs very quickly, as evidenced by our test.

Each member of the team contributed to the project. The work was split between the matrix and elts modules, parsing the data, and the simplex algorithm. Andy and Luis implemented most of the matrix module and parsing function, along with the general framework of the program that allowed us to efficiently run and test our code. Luis, Jason, and Ding worked on implementing and debugging the simplex algorithm. Ding focused on comprehending every step of the algorithm as presented in the Algorithms book and then presenting that information in a more comprehensive manner to the group, while Jason focused on understanding the process behind simplex and on coming up with example problems by hand. Jason and Ding worked on testing simplex, and Luis and Andy provided much of the functionality that allowed for easy testing.

4 Reflection

Our original planning went pretty well and we accomplished most of what we expected to accomplish. We mostly hit both our milestones, though we missed our second milestone by a day. We were definitely surprised by how difficult simplex ended up being, and there were times when we thought we were essentially done with the algorithm and it turned out that we were only halfway there.

Given more time, we would like to polish up input and outputs of our algorithm, including a more sophisticated parsing function and a user friendly output. Within the matrix module itself, there are a few functions that we would have liked to implement including finding the eigenvectors/eigenvalues and calculating the norm of a matrix, among others.

We made the right choice when we decided to drop some of our unnecessary matrix module functions to spend more time on simplex, but we made an early mistake of not bothering to understand simplex until the latter half of the project, and rather than understanding the whole algorithm before we started coding, we chose to only understand the steps we were directly implementing, leading to the restructuring of major portions of our code towards the end. The next time we tackle an algorithm-centric project like this, we should definitely plan out the entire algorithm before we code (including all the possible edge cases that might pop up).

The most important thing we learned is how important good style, readability, and careful planning is to programming and debugging. There were many times when we were confused by what we wrote when we went back to debug our code, and since no one was able to fully internalize every part of the algorithm, it was important to abstract over helper functions and different parts of the code. Having had someone who understood the algorithm entirely (or who had maybe implemented something on this scale before), would have been extremely helpful as it would have allowed us to better plan our time and resources as well as abstract away many of the more detailed concepts.

5 Advice for Future Students

Start as early as possible and put some serious work in. Don't leave the video until the last minute. This is a big project and showing it off is a very rewarding part of it, especially when done well. Also try to understand the general outline for every step before serious implementation, even if it means delaying the implementation. Designing the framework is crucial and it makes the whole process easier to get right the first time. Make sure to write test functions as you write functions and to test each function right after you finish writing it. This way, you will avoid having to hunt down a small bug in one obscure part of your code. You'll know exactly when and where your code went wrong.

Additionally, make sure your group members don't play too much League of Legends or

Starcraft!

But lastly—enjoy the project and be proud of what you've accomplished!