CS 480 **Computational Theory**

Sharpness Exploration

Cole McAllister[2] Scroggs, and Benjamin Walker *May 25, 2015*

Introduction

The goal of Sharpness was for us to learn how to be sharp in creating a *non-trivial* Turing Machine. We initially decided that speed was a more important feature to the machine than ease of implementation or simplicity of the design of the machine. We absorbed complexity to gain efficiency and robustness. After creating a base 2 version, we optimized and added extra features such as simple input error handling and different bases. This report mainly focus on our binary Turing Machine.

Instructions

We decided to use a 3-Tape Turing Machine. Below are descriptions of what each tape is for:

- Tape A is the first factor (multiplier) of the multiplication problem. It accepts any binary number $\{w \mid w \in \{0,1\}^*\}$.
- Tape B is the second factor (multiplicand) of the multiplication problem. Like Tape A, Tape B also accepts any binary number $\{w \mid w \in \{0,1\}^*\}$.
- Tape C is the product. It should be left empty, but can handle any binary number $\{w \mid w \in \{0,1\}^*\}$ if accidentally entered.

Note: Normally the multiplicand is the first value and the multiplier is the second value in a multiplication problem, but for our machine we swapped the two factors positions to make it easier to see the cause of any changes made to the product during each step.

Description of Machine

Our machine has three main phases: the preparation phase, multiplication phase, and cleanup phase.

Note: When a statement "Move Tape X to right" is encountered, it means move the tape head to the right. Ex. ...01**[0]**10... move right becomes ...010**[1]**0...

1. The preparation stage has four main steps:
   (a) Clears any characters from Tape C
   (b) Erases any leading 0's from Tape A and Tape B
   (c) Moves Tape A and Tape B to the rightmost non-empty character
   (d) Checks to see if Tape A or Tape B equals 0. If so writes a 0 and moves to the final step (3.b), otherwise enters the multiplication phase
2. The multiplication stage has three main steps and loops until the end of Tape A is reached:
   (a) Checks if at end of Tape A. If so, move to Cleanup step, otherwise move to next multiplication step
   (b) If Tape A is on a 0, moves Tape A left and Tape B right and a then a 0 is written to the new Tape B position. Then returns to prior multiplication step (2.a)

(c) If Tape A is on a 1, while Tape B is on a 1 or 0:

i. Adds values on Tape B to Tape C (a description of this step will follow after the description of the machine)

ii. Moves Tape B and Tape C to rightmost non-empty character

iii. Moves Tape A left and Tape B right and a then a 0 is written to the new Tape B position. Then returns to first multiplication step (2.a)

3. The cleanup phase has two steps:

   (a) Moves Tape A to the rightmost non-empty character, and for each 1 or 0 encountered, erases a 0 that was added to Tape B and then moves Tape B left

   (b) Final step. All math and cleanup is completed. The value of Tape C is the product of the values of Tape A and Tape B. The input is accepted.

Our binary multiplier machine implements the concept of the carry bit using two states, one state for when there is no carry and one for when there is a carry (Note: unlike base 10, if there is any carry the value is always 1). The following is a description of the process for adding the value of Tape B to the value of Tape C:

1. Starting with Tape B and Tape C on their rightmost non-empty character, add the two characters together. Write the result to Tape C and move to the appropriate state based on the carry value
2. Move Tape B and Tape C right
3. While Tape B is on a 1 or 0:

   (a) add the characters on Tape B and Tape C together, and if in the carry state add an extra 1. Write the result to Tape C and move to the appropriate state based on the carry value

   (b) Move Tape B and Tape C right

4. If in the carry state, write a 1 on Tape C
5. Done adding value of Tape B to Tape C

Assumptions

The following are the assumptions that we made when creating our Turing Machine:

- Binary is easier to implement than other higher base implementations. Unary is easier to implement, but lacks the speed we desired.
- Three tapes are more efficient than one or two tapes. This gives us one tape to store each part of the problem. It allows us to do our math without having to scan left and right for the other numbers. Also for our implementation we also needed the numbers to be able to grow in length, which is far easier on a three tape machine.
- The user will only enter numbers from the set $\{w \mid w \in \{0,1\}^*\}$.
- An empty tape is equivalent to a 0.

Extra Research

Optimizations:

- Input error handling by erasing any content on Tape C before starting.
- Erase leading zeros to remove unneeded steps in our multiplication phase.
- Check if any tapes are empty, and if so skip to finish.
- Remove extra transitions and states from the multiplication stage, shaving off a few steps per multiplication step.
- Cleanup at the end removing all zeros placed on Tape B during the multiplication phase.
- One optimization that JFLAP did not allow us to implement was a NDTM which would have branched and swapped Tape A and Tape B on one branch before continuing with the multiplication. Doing so could have increased the speed of the Turing Machine dramatically depending on the lengths of Tape A and Tape B. This is because of the way that the Russian Peasant algorithm functions. A smaller multiplier results in fewer division steps and in overall increase of speed.

Covering Our Bases:

As we stated in our introduction, speed of the Turing Machine was our primary focus as opposed to ease of implementation and complexity of the machine. We chose to implement binary because it was faster than unary, however we avoided larger bases due to the vast amount of time required to implement them. After we built our binary machine, we realized that our algorithm could easily be generalized for other bases.

JFLAP uses a .jff file which is a simple XML structured file format that is used to store states and transitions for Turing Machines. We found that this format was very easy to work with and that we could create a program to generate Turing Machines automatically for many bases. Scroggs created a C++ program to apply our generalized algorithm to generate Turing Machines in a user prompted base.

What are Multiplication and Division:

In our initial team meeting we decided that the Russian Peasant algorithm would be the best way to implement multiplication in our binary Turing Machine. The Russian Peasant algorithm is generally implemented in base 10, and works by repeatedly doubling the multiplicand and halving the multiplier. Whenever there is a remainder after the multiplier is halved, the multiplicand is added to the product. Once the multiplier is zero, the multiplication stops. We knew that this could be implemented in base two, and is used in computer architecture for efficient multiplication of small numbers. Knowing this could be used in two bases led us to explore if it could be used in other bases as well.

As we delved into the application of the Russian Peasant algorithm in multiple bases we discovered how multiplication and division really work. We will start with the division algorithm:

$$\begin{aligned} a &= q * b + r \\ \frac{a}{q} &= b + \frac{r}{q} \end{aligned}$$

We can apply the division algorithm to prove how the Russian Peasant algorithm works and how it can be generalized.

$$
\begin{aligned}
c &= a_n * b_n \\
c &= \frac{k * a_n * b_n}{k} \\
c &= (k * a_n) * \frac{b_n}{k} \\
c &= (k * a_n) * (\lfloor \frac{b_n}{k} \rfloor + \frac{b_n \bmod k}{k}) \\
c &= (k * a_n * \lfloor \frac{b_1}{k} \rfloor) + (\frac{k * a_n * b_n \bmod k}{k}) \\
c &= (k * a_n * \lfloor \frac{b_n}{k} \rfloor) + a_n * b_n \bmod k \\
c &= (a_{n+1} * b_{n+1}) + a_n * b_n \bmod k
\end{aligned}
$$

Through repeated multiplications of a and divisions of b by k, one can reduce any multiplication to the very simple problem of multiplying a by 0 and adding the remainders of the prior divisions times their respective a_n value together. This is useful because if you only know how to multiply a number by numbers less than or equal to a certain number (perhaps 2), then by setting k to that number allows you to carry out that multiplication by preforming a set of additions and simple multiplications. The Russian Peasant algorithm works by setting $k = 2$.

One can see that it need not be limited only to $k = 2$, but rather can be applied for all numbers $k \geq 2$. This is the foundation of our generalized algorithm. However the main difficulty we encountered was performing the operations in a simple manner. In binary the process is simple. The multiplier is shifted right and the multiplicand is shifted left (logical shift). In other bases, neither operation is simple. We realized that the simplicity in binary resulted because we changed the numbers by the base. If we set k equal to the base then the process simplified to simple shifting as in the binary example for all bases greater than 1. We discovered that we were taught the same method in elementary school (with $k = 10$), but not why it works. Through this exploration we deepened our understanding of multiplication and division.

Conclusion

We set out to surpass all other groups. We created a binary multiplication Turing Machine that could multiply two binary numbers. It had rudimentary input validation and correction and could multiply two numbers efficiently and correctly. We found this task trivial, and thus explored deeper into the subject. We generalized our algorithm to work in other bases and wrote a program to create Turing Machines based on the generalized algorithm. We also delved into the concepts of multiplication, division, and deepened our understanding of the core concepts of the division algorithm and the general process of multiplication. We almost met every day from the time we had our group till it was completed. Among the most useful things we learned is taking initial time to discuss the design of the exploration as a whole, and then verbally pounding out all of it's flaws and pounding in improvements as a group, had many rewards when implementation came about after sleeping on those things for a day. This exploration was very satisfying indeed.