Jason Allen  
Nathan Henninger  
Matt Weaver  
CS 361 – Dr. Sherman

**P2: Turing Machines**

This project consisted of two portions: building a Turing machine simulator and creating a so-called “Busy Beaver” machine that maximizes the sum of non-zeros to an infinite 2-way tape, given blank input before halting.

To build and run the project, perform the following commands from the directory with the source code:

$ javac TuringSimulator.java  
$ java TuringSimulator filename  
(where filename is the name of the file describing the Turing machine to run and its initial input)

***Included files***

Machine.java - Contains the source code to building a Turing machine from an input file.

State.java - Contains the source code to building a State for use by the Machine.

Transition.java - Contains the source code to building a Transition for use by the State.

Tape.java - Contains the source code to represent the Tape for use by Machine.java.

bb.txt - Contains our 4 state Busy Beaver Turing machine.

README - The file you are currently reading. Included for documentation purposes.

First, we will discuss the design of our Turing machine simulator, as well as the ways we tested it and increased its speed.

***Design of the TM simulator***

This program consists of five classes:

- Machine.java - contains the source code to building a Turing machine from an input file

- State.java - contains the source code to building a State for use by the Machine

- Transition.java - contains the source code to building a Transition for use by the State

- Tape.java - contains the source code to represent the Tape for use by Machine.java  
- TuringSimulator.java – contains the main method that creates a machine and runs it

The program was designed from the top down. The sample input gave a visual demonstration showing which lines of input correspond to instance data. Then we decided which of the seven-tuple items required their own object and which would fit in primitive types.

Ultimately, the Machine class contains the meat of the project. It contains an array of states, the size of which is determined by the top line of the input file. These states are numbered from 0 to (top line – 1). Each state holds a list of transitions from it for every input.

These transitions consist of an input (based on the character read from the tape), a nextState (where we’re heading based on the input), a writeSymbol (what we’re writing to the tape prior to moving) and a direction to move (right or left). Cumulatively, the state transitions describe every possible move of the TM based on the input.

A Tape class was used to keep track of the current cell as well as the first and last cell, which are used to check if the TM runs off the bounds of the known values. If it does, a new cell is created with a blank mark to accommodate this. The tape operates as a simple linked list, with the ability to move left or right as called for by the transitions of each state of the TM.

To put it all together, the Machine class contains a run() method that does the following:

1. Checks if the current is the halting state (initially setting it equal to the start state). If not, execute the next transition for the current state.  
2. Write the appropriate symbol for the transition to the tape.  
3. Move right or left, creating appropriate blank spaces on both ends of the tape if the boundaries are reached via the Tape class.  
4. Set the next transition for the new state and repeat the algorithm.

This completes, of course, when the current state is set to halt, which is determined during the parsing of the file.

***Testing the TM simulator***

Testing the machine occurred during several steps. First, we tested our parser by parsing the example input files and making sure the resulting Machine object actually matched the structure described in the file.

Then, an input file matching the one given on Blackboard was created and stepped through to ensure the simulator worked properly and actually halted.

We then manually simulated a few simple Turing machines, and simulated them with our program, to make sure the output was correct.

***Optimizing the TM simulator***

Once we had a functioning Turing machine simulator, we began work on optimizing it, since it was pretty slow (our best Busy Beaver machine took almost 20 minutes to run). To bring this run time down to a more reasonable level, we looked at where the speed bottleneck was happening, which was outputting the contents of the tape. We realized that this output was actually happening while the Turing machine was being simulated, rather than after it finished, which was not only slow, but also was likely giving us extra and incorrect output. By simply waiting to output the tape contents until after the simulation finishes, we were able to reduce the run time from 20 minutes to about 6 seconds (!!!).

Looking through the code, we also saw a few opportunities to make the simulator more efficient. The biggest inefficiency we noticed was that we were using an ArrayList<Integer> for our tape. Since we simulated the tape by inserting new tape cells at the beginning and end of the ArrayList, the whole tape needed shifted in order to move the tape head left, in some cases, and occasional ArrayList resizes meant lots of data was moving around unnecessarily. While most tape operations were O(1), every once in a while, they’d be O(n). To speed this up, we implemented the tape as a simple linked list of ints (simple, because there’s never any need to insert new tape cells anywhere except the beginning and end of the linked list, and deletions never happen). By switching to this implementation, we were able to reduce every tape operation, except printing the tape contents, to O(1).

Also, but using primitive ints, rather than Integers, we were able to avoid the potential overhead from the JVM constantly wrapping/unwrapping Integer objects. Similarly, by switching from using primitive chars, rather than String objects, from some data, we found small gains in performance.

With these optimizations, we were able to reduce our 6 second run time by nearly half to about 3.5 seconds.