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, continue by executing 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. Perhaps the best example for this was creating a file that matched the TM on page 172 of Sipser’s *Theory of Computation*, modified for our file format, of course. We then stepped through this and ensured that the tape, state, and position matched the example at the bottom of the page.

***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 Integer objects, we were able to avoid the potential overhead from the JVM constantly wrapping/unwrapping Integers. Similarly, by switching to primitive chars, rather than String objects, from some data, we found some 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.

***Building our Busy Beaver***

To begin, we wrote a method that could detect if a randomly generated Busy Beaver had a decent chance of halting. Obviously, we couldn’t write something that made certain it halted, since the halting problem is actually impossible. The criteria for this “guess” was:

1. Is there a transition from at least one state to the halting state? If so, is there at least one transition to *that* earlier state? And so on, eventually rejecting the randomly generated TM if this tracing could not bring us back to the start state.

2. Are the input symbols for all of these transitions actually being written somewhere? This step was actually a bit complicated, since two transitions from one state to another means only one of them has to be written for it to be theoretically possible for the machine to halt.

We also tried to detect useless (non-visited) states before looking around online and realizing that this problem was undecideable. Hopefully if that shows up on the test we’ll be ready for it.

This method was quite crude and imperfect, but it gave us a starting point for a search algorithm. From here, if the machine took over 1 billion steps (movements along the tape), we rejected it and tried the next machine. The random Busy Beaver file creator simply used the Java Random class to create transitions between States 0-3 and write them in the appropriate file format. These were then tested by the method described above and run, eventually being rejected if they didn’t halt in a billion steps (a little under a minute on our simulator).

Unfortunately, the best file created by this method had only a score of 5412 after a reasonable amount of time, and we knew it was possible to do much, much better. With our understanding of appropriate search algorithms for this problem low and the clock ticking to the deadline, we decided to abandon this approach.

Instead, since the Busy Beaver problem described in this project is really just a variant of the classic Busy Beaver problem, we decided to look at existing solutions to that problem for ideas, so we could try to hand design a machine that would produce a high score. Since the classic Busy Beaver problem strives to put as many non-blank/zero symbols on the tape as possible, we decided to use a solution to this problem as a start, and then adapt it to our purposes (to maximize totalScore). We ended up settling on the 3-state Turing machine found on <http://www.drb.insel.de/~heiner/BB/simLig34_a.html>, since it’s structure and behavior was reasonably easy to understand, and it simulated quickly enough to be usable. In its original form, that machine gave us a totalScore of 22103, which was pretty good. By modifying the transitions in that machine to produce more 3s, we were able to increase the totalScore to a much more impressive 47189. This change required quite a bit of manual tinkering, along with drawing ideas out on scratch paper to see how we could essentially follow the same path while writing more valuable symbols to the tape.

***Group contributions***

Jason Allen:

Completed the parsing and the initial (but slow) simulator. Created the search algorithm that kind of failed, but performed research and found a good machine online and helped to modify it to make it better.

Nathan Henninger:

Matt Weaver:

Optimized the simulator (e.g., implemented the linked list tape), fixed bugs, and helped with documentation.