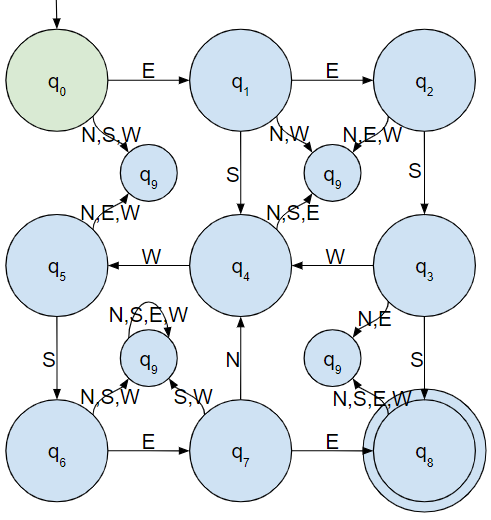
Jon Rippe | Automata HW #2

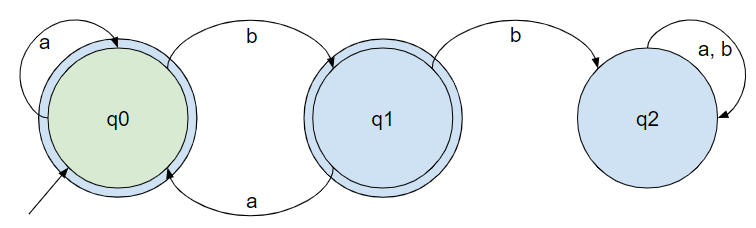
# 1

* Possible routes from box A1 to C3, just by looking at it:
* Possible routes from q­4:
* Create equations & solve:
* DFA recognizing *L*:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |

# 2

# 3

* DFA accepting *F*:

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |

* Regular Expression describing F:

# 4

# 5

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |

# 6

|  |  |  |  |
| --- | --- | --- | --- |
| State | Tape 1 | Tape 2 | Tape 3 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# 7

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |

# 8

|  |  |  |  |
| --- | --- | --- | --- |
| State | Tape 1 | Tape 2 | Tape 3 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# 9

* Per the Church-Turing Thesis, anything that can be written in a coding language (e.g., Java, C++, pseudo-code) can be converted into a Turing machine. Using this Theorem, we can show that subtraction and Euclidean division via Turing machine is possible by writing code that performs the same operation:
  + Subtraction is so much like addition that I could likely implement it with a Turing machine no more complicated than my shift-addition multiplication machine in #7. A machine capable of subtraction would work very similar to the addition machine described in #5 and would only require a few extra states that convert the second tape to its 2’s complement before performing an addition operation:

int subtract(int num1, int num2){  
 num2 = get2sCompliment(num2);  
 answer = num1 + num2;  
 return answer;  
}

While (tape2 != □): //need a state for this operation  
 flip bit  
 move tape2 Right  
move tape2 Left

//need 2 states for this operation (one for carry, one for no carry)  
carry = add 1 to tape2 value  
While (tape2 != □):  
 move tape2 Left  
 if(carry):  
 carry = add 1 to tape2 value  
move tape2 Right

//We now have an addition problem that can be solved by our defined machine.

* + Using the above algorithm for subtraction, we can create a new Euclidean division algorithm:

Tuple euclDivide(int num1, int num2){  
 quotient = 0;  
 While(num1 > num2){  
 quotient++;  
 num1 -= num2;  
 }  
 remainder = num1;  
 return (quotient, remainder);  
}

# 10

* Using the same argument above, we can argue that any code can be converted into a Turing machine, and multiple Turing machines can be represented by a single Turing machine. The following code specifically performs the operation using the logic from our defined Turing machines for addition and multiplication:

tape1 = a;  
tape2 = b;  
tape3 = c;  
tape4 = d;  
tape5 = TMadd(TMmult(tape1, tape2), TMmult(tape3, tape4));  
tape6 = TMmult(tape2, tape4);

* For a simplified version :

tape1 = a;  
tape2 = b;  
tape3 = c;  
tape4 = d;  
tape5 = TMmult(tape1, tape3);  
tape6 = TMmult(tape2, tape4);

# 11

In class, we showed that each Turing machine can be mapped to an integer number through a function , meaning for each Turing machine a unique Gödel Number.

Corollary: Because the size of is greater than the size of and the number of Turing Machines is equal to the size of , the size of must be greater than the number of Turing machines, and there are more real numbers than Turing machines.

# 12

First, let’s see if we can keep track of both and by changing the way we do the summation. Because addition is commutative, we can add down from a number to instead of adding up. This is what we’ll do, so now our first tape starts at and decrements after each summation step. When the Turing machine goes to check the first tape and finds it to be , it terminates in the accept state.

Next, we’re going to get rid of all the fractions inside the parentheses by finding a common denominator and rewrite using algebra. I’m not actually going to do this because it would be a nightmare, but if I *had* to build this machine, it would be my next step.

Now our machine works with the numerators of Bellard’s formula. It uses addition, subtraction, and multiplication to find the numerator inside the parentheses and store it in tape #2. It then uses multiplication, or another algorithm for determining if our current is odd/even, to get the other numerator within the summation, multiply it with tape #2, overwriting tape #2’s contents. Our machine then does the same with the denominators, storing the resulting values in tape #3, performing an extra calculation for the denominator outside the summation.

This continues until the value stored in tape #1 is .

A quick glance at Bellard’s formula shows that the denominator grows at a maximum rate of while the numerator remains the same (aside from alternating positive to negative). This mean that our discrepancy from true should be as will shrink at a faster rate than .