William McLaughlin

Problem statement

A deterministic finite automaton (DFA) is a 5-tuple $(Q, \Sigma, \delta, q0, F)$, where

- Q is a finite set called states
- Σ is a finite set called the alphabet
- $\delta : Q \times \Sigma \rightarrow Q$ is the transition function
- $q0 \in Q$ is the start state, and
- $F \subseteq Q$ is the set of accepting states.

A DFA is responsible for taking a string as input, evaluating it, and determining if it's an accepting or rejecting string. It does this evaluation in constant O(n) time. Every state has a transition on every input, and only one. The transitions between these states can be demonstrated in a transition table. DFAs recognize exactly the set of regular languages.

The problem I sought to solve was adding two binary strings, in Dragon form. Dragon form is when each string reverses it's parity on every other 1 bit in the string. For example, 0111 in binary is $+ 0\times2^3 + 1\times2^2 + 1\times2^1 + 1\times2^0 = 4+2+1 = 7$. In dragon form, this becomes $+ 0\times2^3 + 1\times2^2 - 1\times2^1 + 1\times2^0 = 4-2+1 = 3$ in decimal. The solution in Dragon form can be written in two different ways: 0101 or 0111, which is 4-1 and 4-2+1 respectively. Many strings in Dragon form have two representations, one with an odd number of 1s and one with an even number of 1s.

To solve this problem required the construction of three DFAs and four general steps. The first three steps, each with their own respective DFA, were as follows:

1. Split both of the input strings into two
2. Get the two's compliment of the second string of the split
3. Add all four strings together
4. Convert the output to Dragon

Here is a table of decimal to binary to dragon conversions 0-9

DECIMAL	BINARY	EVEN 1 DRAGON	ODD 1 DRAGON
0	0000	0000 0000	-
1	0001	0000 0011	0000 0001
2	0010	0000 0110	0000 0010
3	0011	0000 0101	0000 0111
4	0100	0000 1100	0000 0100
5	0101	0000 1111	0000 1101
6	0110	0000 1010	0000 1110
7	0111	0000 1001	-
8	1000	0001 1000	0000 1000
9	1001	0001 1110	0000 1001

1. Splitting the Input Strings:

The first step of the process is to split each input string into two. The split is determined by the fist location of a 1 in the string, then alternating on each next occurrence of another 1. The first string generated is the positive values, while the second string is the values to be subtracted. My DFA would wait in q0 until it saw a 1, then alternate between q1 and q2 representing which string to push to. The δ for my DFA is of the following form: δ(q, a, b) where q is the next state and a, b are the values to be pushed to the first and second string respectively.

-	0	1
q0	q0 : 0, 0	q1 : 1, 0
q1	q2 : 0,0	q2, 0,1
q2	q1 : 0,0	q1, 1,0

2. Finding Two's Compliment:

In order to subtract the two second strings from the first, we need to find the Two's Compliment. There is no negative numbers in binary. The δ for my DFA is of the following form: δ(q, a) where q is the next state and a is the value to be pushed to the string. My algorithm works like a binary adder for two strings, except that we always know the other the string is: 0 * (the length of the input – 1) + 1. e.g. if the input is 1101, then the length is 4, we expect 3 0's followed by a 1: 0001. Additionally, since we need to expect the 1's compliment to be completed as well, it will treat every bit as the opposite. The first state will always be adding a 1, while the other two will add 0's, keeping track of any carry. (Which we discard after anyways, so maybe this is wasteful.)

-	0	1
q0	q2 : 0	q1 : 1
q1	q1 : 1	q1 : 0
q2	q2 : 0	q1 : 1

Note: Often when doing 2's Compliment it's important to take note of the most significant bit in the resulting string to determine if the string is negative. In this case, the string can never be negative, so I ignored this case.

3. Adding Four Strings:

When adding two binary strings together, the algorithm must keep track of the carry. In the case of adding four binary strings together, there is the possibility of 3 carries. This would be the case when adding together the following strings: 111, 111, 111, and 111. On the first operation, four 1's yields a 0 with two carries. Next, four 1's and two carries yield another 0 and three carries. Lastly, four more 1's with 3 carries gives us a 1 with two remaining carries.

At this point, my algorithm is done, since we are ignoring the carries in this assignment. If we wanted to keep track of the carries, we could either keep feeding the DFA four 0's, or we could handle the carries programmatically with something like a switch statement. The δ for my DFA is of the following form: δ(q, a,) where q is the next state and a is the value to be pushed onto the string. To simplify my algorithm, I sort the

-	0000	0001	0011	0111	1111
q0	q0 : 0	q0 : 1	q1 : 0	q1 : 1	q2 : 0
q1	q0 : 1	q1 : 0	q1 : 1	q2 : 0	q2 : 1
q2	q1 : 0	q1 : 1	q2 : 0	q2 : 1	q3 : 0
q3	q1 : 1	q2 : 0	q2 : 1	q3 : 0	q3 : 1

input before feeding it in, that way I only have 5 different inputs rather than 16.

4. Convert Result to Dragon

This portion of the project is unfinished. While I was unable to construct a DFA for this process, I did implement an algorithm that I think is part of the way there. My initial misconception was that every dragon form of binary string would flip the bit of highest magnitude + 1 and always keep the next bit as 1. This is true for decimal 1-12, but was disproven at 13.

Here are the steps so far:

1. Take an integer(n) as input and convert it to binary = x
2. Find the highest order of magnitude = a = 2^{n+1} (e.g. 1010: n = 3)
3. Check if the difference n – x and check if it's a power of 2
4. If it is, add that value and x to n: **DONE**
5. If it is not, add n and 2^n to x *(This is where I went wrong, it could be any 2^n, 2^{n-1}... 2^{n-n})*
6. Take the difference between (x – (a – 2^n)) = c and check if that's a power of 2.
7. If it is, add x + a + 2^n: **DONE**
8. If it is not, get the highest order of magnitude of c in binary and start from step 3 again

What I should have done: at step 5, check if any 2^n, 2^{n-1}... 2^{n-n} is a power of 2. If none of them are, pick one and continue, if it fails, go back and pick the next one to move on to step 6.

This formula works for all values binary 1-12.

Sample Input and Output:

Input 1: 0001
Input 2: 1001
Sum in decimal: 0000 1000
Sum in dragon: 0001 1000

Input 1: 0010
Input 2: 1000
Sum in decimal: 0000 1010
Sum in dragon: 0001 1011

Input 1: 0011
Input 2: 0111
Sum in decimal: 0100
Sum in dragon: 0000 1100

Input 1: 0100
Input 2: 0110
Sum in decimal: 0110
Sum in dragon: 0000 1010

Input 1: 0101
Input 2: 0101
Sum in decimal: 0000 1010
Sum in dragon: 0001 1011

Input 1: 0110
Input 2: 0100
Sum in decimal: 0110
Sum in dragon: 0000 1010

Input 1: 0111
Input 2: 0011
Sum in decimal: 0100
Sum in dragon: 0000 1100

Input 1: 1000
Input 2: 0010
Sum in decimal: 0000 1010
Sum in dragon: 0001 1011

Input 1: 1001
Input 2: 0001
Sum in decimal: 0000 1000
Sum in dragon: 0001 1000

Summary and Conclusions

There is a lot more than meets the eye in this project. The initial idea seems fairly simple, but actual implementation proved to be quite challenging. The initial step of the process, splitting the strings into positive and negative halves, was immediately obvious to me. At the time, I thought I had solved the whole problem, which clearly wasn't the case. This algorithm was later described to me as the Dragon. From what I see online, it seems like the Dragon entails some kind of flipping 90^0 while this process seems to invert strings s full 180^0? I'm not really sure.

The most difficult part of this assignment for me was reverting the string to a Dragon string. This process seems like it would require a lot of guessing a checking, which may be perfect for an NFA. Even then, the NFA requires knowing the answer initially to compare it's guesses against. Which doesn't seem very helpful to me.

The last question I have unanswered is why we ignore caries in our addition algorithm. The resulting math checks out, but it seems important to me nonetheless.