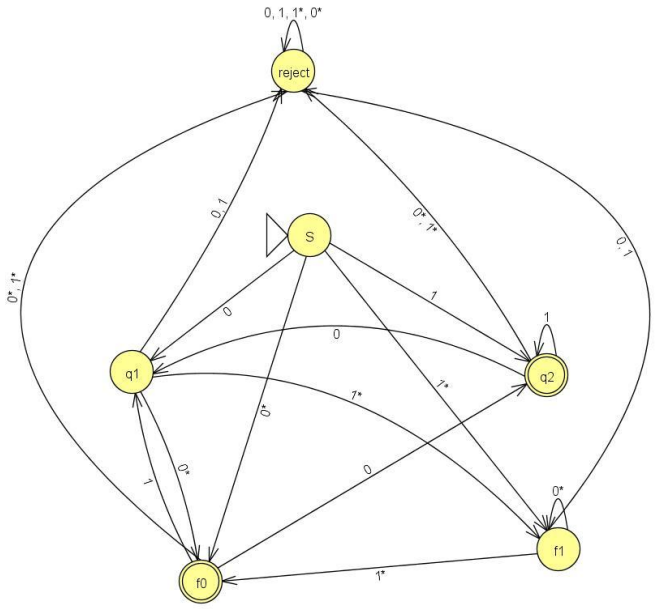
CS 454 Final Project Writeup

By Eric Fleming, Trent Jones, Cody Rimes

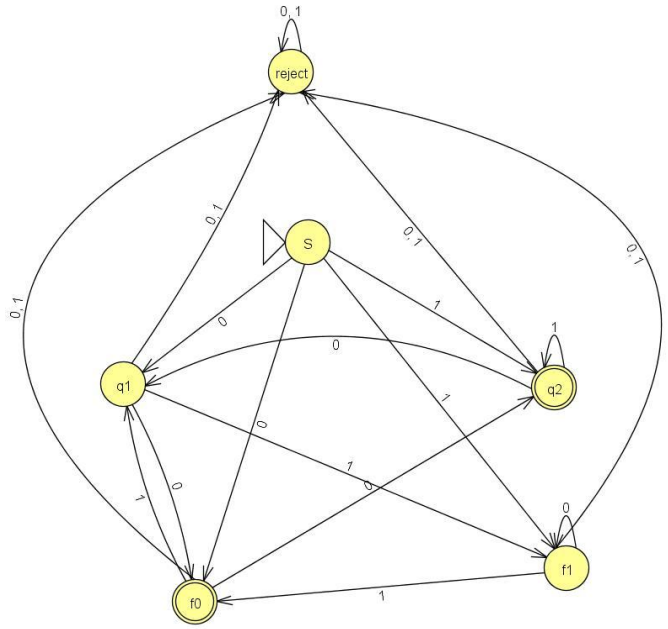
Our group chose to focus on a solution to problem 1A), which is stated as follows: In the lights-out puzzle, you are given a string over the alphabet {0, 1} in which 1 (0) represents a light bulb that is on (off). Each light bulb has a switch next to it. If you press a switch, it toggles the state of the light bulb it is next to, as well as the neighboring light bulbs. For example, if the current pattern is 10010, and you press the third switch, it will result in the pattern 11100. The goal of the puzzle is to turn all the lights on by pressing switches in some sequence. (The original puzzle has a 5 by 5 arrangement of lights, but here we consider the lights in a single row, but of arbitrary length.) The goal of this project is similar to the puzzle, and to solve the following problems:
(a) Define the language
L = {w | w can be turned to a string of 1's by pressing some combination of switches}.
Show that L is regular by building a DFA. Also design an algorithm that uses this DFA to find the sequence of switches that need to be pressed to turn all the lights on.

To solve the first part of this problem, we designed three automata to help us determine if any string of n length consisting of 1's and 0's can be turned into a string of just 1's by flipping switches. Our first DFA takes into account whether a 0 or 1 is the current character, and also contains characters 0* and 1* for flipping an input signified by a star. The DFA is pictured below.
Original DFA:

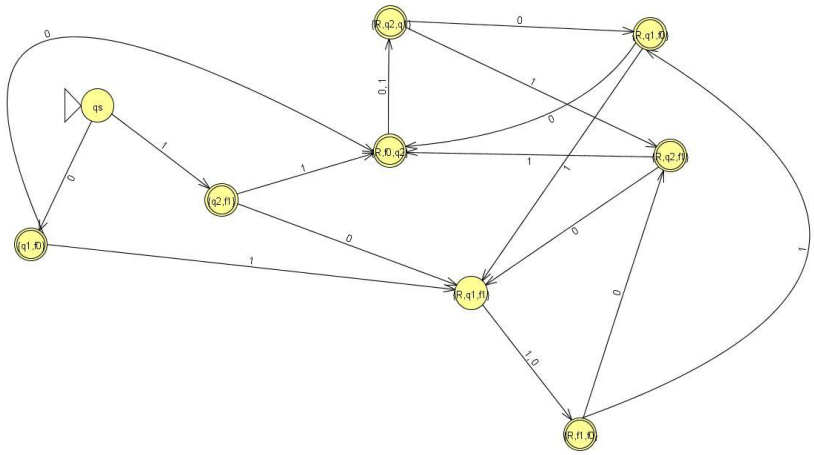


We constructed this initial DFA, to determine what switches to flip to produce a string of only ones. The DFA keeps track of the last symbol, the current symbol, and whether a flip has occurred. Symbols that would result in a substring that is not all 1's cause the DFA to enter the

reject state, while strings that end in one of the accept states are strings that either start with all 1's or have flipped switches in the right order to become a string of all 1's. From this initial DFA we designed an NFA that removes the 0* and 1* steps and replaces them with 0 and 1. This NFA is pictured below.
NFA created from DFA:



Using this NFA, we utilized the lazy subset construction method to convert an NFA to a DFA. We use this final DFA in our project in order to prove membership of a given input string. The final DFA is pictured below. This DFA proves that L is a regular language.
Final DFA:



In order to solve the second part of our problem we designed 3 functions along with a driver function to handle different parts of solving an input string. Our first function is bool validCheck(std::string entry), which takes as input a string of 1's and 0's and returns whether or not it can be converted into a string of just 1's. To start out this function we use a 2D array to simulate our final DFA, with an additional bool array to handle which states are accepting states. We then initialize the starting state by setting our int state to 0, and parse the input string character by character taking transitions based on the current input symbol and the current state. Once all symbols have been parsed we return true if the ending state is an accepting state. Code for this is as follows:

```
while(index < entry.length()) {
        if(entry.at(index) == '1') {
            state = delta[state][1];
        }
        else if(entry.at(index) == '0') {
            state = delta[state][0];
        }
        else {
            std::cout << "Error, unexpected value at space " << index << " character is : " <<
entry.at(index) << std::endl;
            std::cout << "Please only input a string of 0's and 1's" << std::endl;
            exit(1);
        }
        index++;
    }
    //string is parsed return accepting
    return accepting[state];
```

Valid Check's **Time Complexity: O(n)**

Our second function is std::vector<int> flippedPlaces(std::string input), which takes an input string of 0's and 1's and returns a vector that signifies which places should be flipped to turn the input string into a string of all ones. This function initializes a 2D array to hold our first DFA, and values to hold the accepting and reject states. In this function we use a struct to hold our current state, the number of flips, and a vector for which place has been flipped, shown below.

```
struct node {
    int state;
    int flips;
    std::vector<int> fPos;
}
```

We use a queue of these nodes to perform a breadth first search using the input string and our DFA. On every input character we take the current state and adjust it using our DFA creating

two nodes, one where we don't flip the input character and one where we do. In the flipped case we increment our flip counter within the node and mark off the place in our flip vector. We then check to see whether either created node has entered the reject state, ensuring we don't add extra work to the queue to keep our time complexity down. Once the last character has been parsed, we iterate through all of the nodes left in the queue and keep the node with the lowest count that has ended in an accepting state, ensuring that we always return the least number of flips if there is more than one path to a string of all ones. This code is shown below :

```
   std::queue<node> nodes;
   struct node root = { 0, 0 };
   int stringIndex = 0;
   nodes.push(root);
   //iterate through each input, adding the char and their dot to the queue if they don't end in a
reject state
   while(stringIndex < input.length()) {
      int queueSize = nodes.size();
      int queueIndex = 0;
      int currentChar;
      if(input.at(stringIndex) == '1') {
         currentChar = 1;
      }
      else {
         currentChar = 0;
      }
      while(queueIndex < queueSize) {
         // node has state, counter for flips, and vector to determine flip position
         node current = nodes.front();
         nodes.pop();
         node regNode = {flippableDelta[current.state][currentChar]  , current.flips, current.fPos};
         regNode.fPos.push_back(0);
         node flipNode = {flippableDelta[current.state][currentChar + flipOffset]  , current.flips+1,
current.fPos};
         flipNode.fPos.push_back(1);
         if(regNode.state != rejectState) {
            nodes.push(regNode);
         }
         if(flipNode.state != rejectState) {
            nodes.push(flipNode);
         }
         queueIndex++;
      }

      stringIndex++;
   }
```

```
    node winner;
    int leastFlips = input.length()+1;
    while(!nodes.empty()) {
        if((nodes.front().flips < leastFlips) && (nodes.front().state == accept1 || nodes.front().state
== accept2)) {
            winner = nodes.front();
            leastFlips = winner.flips;
        }
        nodes.pop();
    }
```

Flipped Places Time Complexity: O(n)

The last function we use in our program is void flipSimulator(std::string input, std::vector<int> flipPlaces), which creates a visual representation of what our algorithm accomplishes. We pass in the original input string as well as the vector representing which places should be flipped. We print the original string and then iterate through the vector, flipping the marked places according to our lights out rules and printing the new string at every flip until we reach the end of the vector and the string should be all ones.

Flip Simulator Time Complexity: O(n)
Overall Time Complexity: O(3n): O(n)

Data Structures:

2D arrays: Flipped places and valid check both use 2D arrays that simulate the DFA's, with the row representing the current state and the column representing movement on an input.
vectors: We use vectors to store which place we have flipped.
structs: Stores the current state of the string and the amount of flips as well as a vector that keeps track of flip position.
queue: we use a queue to store structs for our breadth first search to find the least amount of flips needed to make the string all 1's.

Example Inputs:
Ctrl c or entering anything other than a string of 0's and 1's will exit the program
Compile with : g++ main.cpp
Run with : ./a.out

Please Enter a string of 0's and 1's:
1000
The string 1000 can be solved!

least amount of flips is 1
original string is 1000
flipping place 2 new string is : 1111
This is the final string!
Please Enter another string

11110
The string 11110 cannot be solved :(
Please Enter another string

11100
The string 11100 can be solved!
least amount of flips is 1
original string is 11100
flipping place 4 new string is : 11111
Please Enter another string

0001000001000
The string 0001000001000 can be solved!
least amount of flips is 9
original string is 0001000001000
flipping place 0 new string is : 1101000001000
flipping place 3 new string is : 1110100001000
flipping place 4 new string is : 1111010001000
flipping place 5 new string is : 1111101001000
flipping place 6 new string is : 1111110101000
flipping place 7 new string is : 1111111011000
flipping place 8 new string is : 1111111100000
flipping place 9 new string is : 1111111111100
flipping place 12 new string is : 1111111111111
This is the final string!
Please Enter another string

Conclusion: The set of strings w in which w can be turned into a string of just ones by flipping switches according to the lights out game is a regular language. In fact most strings composed of 0's and 1's can be turned into a string of 1's by flipping bits. We experimentally determined that strings of length 2, 5 , 8 … (2 + 3n) had permutations that could not be solved by flipping switches, and our final DFA has a cycle that demonstrates this. Strings of any other length are always solvable no matter the order. Our program runs very quickly even when presented with strings of length 200+, and greatly simplifies what is a somewhat difficult problem for a person to do. This project gave us a new respect for the power of automata and their usefulness in solving programming problems.