GA4 - Report
CS 325
Baorong Luo, Joy Jin, and Sam Zink
14 March 2024

## Description of Algorithm(s)

### Preface: Overview of Problem Reduction

This algorithm's switches and lights problem is a boolean satisfiability problem pertaining to whether circuits of lights and switches can have some switch configuration such that all lights can be turned off. Each input circuit is composed of lights with initial states of on or off, and switches with connections to lights. The algorithm reduces this problem to a 2-SAT problem, which it passes to a blackbox 2-SAT problem solver.

In the 2-SAT representation of the problem, each light is represented as two logical clauses, with the states of each light's two connected switches represented as the two literals of the light's logical clauses.

Each light's representation as logical clauses will depend on the light's initial on/off state in the input. The logical clauses representations of lights are described below, with P representing one of the light's connected switches, Q representing the light's other connected switch, and the "~" negation symbol in front of a switch representing that the switch's state is set to opposite its initial state in the input.

- If the light's initial state is on, the light will be represented by the logical clauses: (P or Q) and (~Q or ~P)
- If the light's initial state is off, the light will be represented by the logical clauses: (P or ~Q) and (P~ or Q).

In the 2-SAT representation of this problem, with lights represented as clauses, the clauses of all lights will be joined together by "and" logical operators.

### Step 1: Define Circuit Class

A circuit class is defined to encapsulate switch, light, light state, and switch-light connection data for each instance of a switches and lights problem. The defined methods of this class are detailed below.

**__init__ Method:**

The circuit class defines this method to initialize a new circuit class object; this method requires the below described parameters:

- "switches" parameter:

An integer representing the number of switches in the circuit.
- “lights” parameter:
  An integer representing the number of lights in the circuit.
- “state” parameter:
  An array of m elements, with element values of 1 or 0, representing the initial on/off states of each light.
- “connections” parameter:
  An matrix of up to mn elements, representing the lights connected to each switch.

Accordingly, this method outputs a circuit class object with properties as described below:

- A “switches” property, which is a list of n elements, where n is the number of switches from the “switches” integer parameter. The value of each element is its index number in the list, with the first index being 0 and the last index is (n-1); these index numbers serve as identifiers for the switches.
- A “lights” property, which is a list of m elements, where m is the number of lights from the “lights” integer parameter. Each element is empty and without a value.
- A “state” property, which is a list of m elements, where the value of each element is either 1 or 0 to represent the initial on or off state of the light at the identical index of the “light” property array.
- A “connections" property, which is a matrix (list of lists) of up to mn elements, with n indices on one axis from index 0 to index (n-1) representing each switch, and with up to m elements on the other axis starting from index 0 representing the lights each switch is connected to. The value of each element is either 1 or 0 to represent the initial on or off state of the light at the identical index of the “light” property array.

**Getter Methods:**
The circuit class defines getter methods for all private attributes of the class. These getter methods each take no parameters and can be called for any circuit class object, each method's returns are described below.

- **Get_lights Method:**
  Returns the lights attribute, which is a matrix (list of lists).
- **Get_state Method:**
  Returns the state attribute, which is a list.
- **Get_switches Method:**
  Returns the switches attribute, which is a list.
- **Get_connections Method:**
  Returns connections attribute, which is a matrix (list of lists).

**fill_lights Method:**
This method is responsible for populating the self.lights attribute in the circuit class based on the self.state attribute. It defines the behavior of the lights in the circuit. This function iterates over the elements of self.state, where each element represents the state (either '0' or '1') of a specific component in the circuit. For each element in self.state, it checks whether the state is equal to '1'. If it is, it sets the corresponding entry in self.lights to a specific pattern.

**create_clauses Method:**
This is the function declared as part of the circuit class that can be used to create clauses. This function is part of a larger system for handling logical clauses in a circuit. It modifies the internal representation of lights based on the connections in the circuit.
After calling self.fill_lights(), which initializes or fills the self.lights attribute in the circuit class. The algorithm proceeds to use nested loops to iterate over the connections in the circuit and modify the lights accordingly. It replaces placeholders like 'P' or 'Q' with the actual switch numbers from self.switches.

### Step 2: Call get_circuits() Helper Function

The get_circuits() function reads information from an input file representing two instances of switches and lights problems, and creates two instances of circuit class objects with the input file information. It does this via the mechanisms detailed below.

**Creating First Circuit Class Object “Circuit1”:**
The helper function first opens the input file specified by input_file_path in read mode. In code, it’s input_file = open(input_file_path, "r"), then, it reads the first line of the input file and discards it (input_file.readline()). The second line of the input is then split by commas, converted the elements to integers, and assigned them to switches and lights. The third line is also read, and split by commas, then the resulting list is assigned to the variable ‘state’. After the third line, each represents a connection. The algorithm converts the elements to integers and appends them to the ‘connections’ list. The algorithm then creates an instance of the circuit class named Circuit1 using the collected information.

**Creating Second Circuit Class Object “Circuit2”:**
Using a similar method, the helper function aims to read data for the second circuit. The algorithm reads the first line of the input file and discards it (input_file.readline()). The algorithm reads the second line, splits it by commas, converts the elements to integers, and assigns them to switches and lights. Then, the algorithm reads the third line, splits it by commas, and assigns the resulting list to ‘state’.

Using a similar method as circuit 1, the algorithm reads the next 'switches' lines, each representing a connection. The algorithm converts the elements to integers and appends them to the connections list. The algorithm then creates an instance of the circuit class named Circuit2 using the collected information.

**Cleanup and Return:**
At the end of this step, the algorithm closes the input file (input_file.close()). The two calls to the get_circuits() method in this step return two circuit class objects, Circuit1 and Circuit2 stored in a list named Circuits.

### Step 3: Call create_clauses() Helper Function for Each Circuit Class Object

The create_clauses method defined in the circuit class iterates through the given class object's "lights" property, which was initialized as a list of empty elements, to fill out each element with the appropriate literals to represent the corresponding light's initial state in terms of the possible states of its connected switches. It does this via the mechanisms detailed below.

**Calling the fill_lights Method:**
The create_clauses method first calls another circuit class method, the fill_lights method, for each of the two circuits class objects. The fill_lights method iterates through the given class object's "lights" property, which was initialized as a list of m empty elements in Step 2 , assigning each element literal values in terms of variables P and Q depending on the corresponding light's initial on/off state data found in the class object's "state" property list.

For each element in the class object's "lights" property list, if the element at the identical index in the class object's "state" property list has:

- A value of 1, the light being represented had an initial state of "on" in the input. As such, only one of the light's connected switches must change states for the light to have the desired state of "off".
  This can be expressed in a CNF format as the clauses "(P or Q) and (~P or ~Q)", where P and Q are the initial states of the light's connected switches, and where ~P and ~Q are the negation of the initial states of the light's connected switches.
  To match the input format of the 2-SAT solving blackbox linked in the assignment description, the element in the class object's "lights" property list will be given the value ["P","Q"] , ["~P","~Q"] to represent the clauses.
- A value of 0, the light being represented had an initial state of "off" in the input. As such, either both or neither of the light's connected switches must change states for the light to retain the desired state of "off".
  This can be expressed in a CNF format as "(P or ~Q) and (~P or Q)", where P and Q are the initial states of the light's connected switches,

and  where ~P and ~Q are the negation of the initial states of the light's connected switches.
To match the input format of the 2-SAT solving blackbox linked in the assignment description, the element in the class object's "lights" property list will be given the value ["P","~Q"] , ["~P","Q"] to represent the clauses.

**Replacing Each Light's P and Q Variables with Switch Identifiers:**
Now that each light is appropriately represented in the class object's "lights" property matrix as clauses based on its initial on/off state, in terms of P and Q variables, this step iterates through the class object's "connection" property matrix. In this "connection" property matrix, each switch is represented at the identical index of the matrix's outer list (of its list of lists) as a list of connected light index numbers.

For each switch, this step iterates through its list of connected light index numbers and does the following:

> For each light in a switch's list of connected light index numbers, this step indexes into the outer list of the class object's "lights" property matrix (list of lists) to access the corresponding light's CNF clause representation.
>
> If the clause representation still contains the variable "P", instances of "P" in the light's clause representation are replaced with the index of a switch connected to the light, which is the index of the switch's list representation in the "connection" property matrix's outer list wherein the light index being accessed was found.
>
> Else if the clause representation no longer contains the variable "P", instances of "Q" in the light's clause representation are replaced with the index of a switch connected to the light, which is the index of the switch's list representation in the "connection" property matrix's outer list wherein the light index being accessed was found.

After iterating through all elements of the "connection" property matrix to update clause representations in the "lights" property matrix, all instances of variables "P" and "Q" should be replaced by switch index numbers.

Below please find an example:

A switch is represented at index 2 of the outer list of the "connection" property matrix, and as a list of light index numbers containing light index number 3

If the clause representation of the light at index number 3 in the outer list of the "lights" property matrix is ["P","Q"] , ["~P","~Q"] , instances of "P" in this

representation would be replaced with the switch's index number, 3, for an updated clause representation of ["3","Q"] , ["~3","~Q"].

**Return:**
At the end of this step, each Circuit object's self.lights attribute is populated with clause representations of each light, in terms of each light's connected switches as literals.

### Step 4: Create two_cnf Class Objects for Each Circuit Class Object

The two_cnf class is predefined in the outside code linked in the assignment description at "https://github.com/arunptl100/SAT-Solver/blob/master/sat-solver.py" for use as a blackbox that can solve the 2-SAT reduction. Since this is to be treated as a blackbox per the assignment description, the two_cnf class properties and methods will not be described in detail in our report; the two_cnf class properties and methods will only be referenced to the extent that our algorithm and passes arguments to class methods, receives outputs from class methods, and accesses properties of two_cnf class objects.

**Initialize two_cnf Class Objects for Each Circuit:**
Initialize a two_cnf class object for each circuit problem, two_cnf class object "formula1" to hold the 2-SAT reduction of the input file's first circuit problem and two_cnf class object "formula2" to hold the 2-SAT reduction of the input file's second circuit problem.

Each two_cnf class object is initialized using the predefined __init__ method of the two_cnf class with no arguments passed to the method. Each call to the predefined __init__ method outputs a two_cnf class object containing a "con" property, which is initialized as an empty list.

**Add Clauses to the two_cnf Class Objects for Each Circuit:**
Iterate through the outer list of each Circuit class object's "lights" property matrix to access clause representations of each light.
For each index of this outer list of the Circuit class object's "lights" matrix:

- The index of the outer list of the Circuit class object's "lights" matrix now holds a matrix (list of lists) representation of the represented light's two clauses in terms of switch index literals from Step 3.

  The represented light's two clause representations are each a list of two literals and each clause can be individually accessed by further indexing into the matrix held at the current index of the outer list.

  Add each of the two clauses to the two_cnf class object corresponding to the Circuit class object to which the clauses belong.

This is done by calling the predefined add_clause method of the backbox's two_cnf class from the appropriate two_cnf class object and passing the clause representation as an argument to this add_clause method.

**Return:**
At the end of this step, each two_cnf object's self.con attribute is populated with all clauses representing the two_cnf object's corresponding Circuit class object.

### Step 5: Use Blackbox 2-SAT Solver

The two_sat_solver() method is predefined as part of the predefined two_cnf class of the outside code linked in the assignment description for use as a blackbox that can solve the 2-SAT reduction.

**Initialize Variables:**
Initialize two variables to hold the result of each 2-SAT problem, a variable named f1_answer for the first circuit problem's corresponding two_sat object and a variable named f2_answer for the second circuit problem's corresponding two_sat object.

**Call the two_sat_solver Method for Each two_cnf Class Object:**
Call the predefined two_sat_solver method for each of the two_cnf class objects. Pass the "formula1" two_cnf object holding reduction of the first circuit problem as an argument to the first call, and store the return value in the f1_answer variable. Pass the "formula2" two_cnf object holding reduction of the second circuit problem as an argument to the second call, and store the return value in the f2_answer variable.

### Step 6: Write Results to Output File

Open the output file (output_file_path) in write mode.
Write the results of the 2-CNF formula solutions for both circuits to the output file, with a "\n" character after each result to match the output format specified in the assignment description and test files.
Close the output file.

## Running Time Analysis of Reduction to 2-SAT Problem

### Running Times of Circuit Class Methods Used in Our Reduction

Each call to the __init__ method is completed in O(m+n) time, because:

- The input “state” list and the input “connections” matrix do not need to be modified and are assigned to the new class object’s respective state and connections properties in O(1) time.
  *Note: Confirmed with John during his 3/13/24 office hours that passing a list or a matrix to a class property without modifying is O(1).*
- A list is initialized for the new class object’s “switches” property in O(1) time. For each of the n switches represented by the input integer n, an element is added to the “switches” property list in O(1) time per element; n elements are added to the list in O(n) time.
- A list is initialized for the new class object’s lights property in O(1) time. For each of the m lights represented by the input integer m, an element is added to the “lights” property list in O(1) time per element; for m lights, m elements are added to the list in O(m) time.
- T(__init__) is the sum of the run time of the above described mechanisms of __init__, that is:

  $$T(\text{__init__}) = O(1) + O(1) + O(n) + O(m)$$

  The O(1) time complexity terms to store inputs in the “state” and “connections” object properties are dropped in the big O representation of __init__’s time complexity, since lesser terms and constant coefficients are dropped in big O notation.

  This leaves us with O(m+n) as __init__’s big O time complexity, so __init__ = O(m+n).

Each call to the fill_lights method is completed in O(m) time, as for each of the m lights the following constant number of operations are performed:

- An “if” statement is evaluated in O(1) time.
- One of two list values are assigned to an element of the class object’s “lights” property list in O(1) time.

Each call to the get_state method is completed in O(1) time, as this method returns the class object’s existing state property, which is a list.
Each call to the get_switches method is completed in O(1) time, as this method returns the class object’s existing switches property, which is a list.
Each call to the get_connections method is completed in O(1) time, as this method returns the class object’s existing connections property, which is a matrix.

**Running Time of Our Reduction to a 2-SAT Problem**

In Step 1, the circuit class is defined in O(1) time, since a constant number of class methods and properties are defined; Step 1 = O(1)

In Step 2, the input file is processed into two circuit class objects, in O(nm) time for each circuit class object. This is because for each instance of a lights and switches problem in the input:

- The first line of the input instance is read and stored as a list of two elements in O(1) time. This is because each of the two integers, n and m, must be within the given range: $1 \leq n, m \leq 1000$. As such, the number of characters between n, m, and the comma separating them must be less than or equal to some constant number of characters.
- Two variables, “switches” and “lights” are initialized and assigned their respective values from the first line list elements in O(1) time.
- The second line of the input instance is read and stored as a list in O(m) time, since there are up to some constant number of characters for each of the m lights represented on the second line.
- The remaining n lines of the instance are read, with the values from each line stored in a list, and the lists of each line’s values stored at the n indices of an outer list to comprise a matrix. This is done in O(nm) because:
    - Each of the n lines represents a switch and contains a light index value for each light connection to the switch. There are up to m lights connected to each switch. As such, their number of light index values found in all n lines is upper bounded by nm.
    - Each value has up to some constant number of characters, since the index number of lights are within the range $1 \leq m \leq 1000$, so each value can be read in O(1) time.
- The objects created by the above described mechanisms for each instance of input lights and switches are passed to the __init__ class method as arguments. As described above, the __init__ class method runs in O(m+n) time.
- Lesser terms are dropped in the big O representation of time complexity, so the greatest time complexity term of the above described mechanisms, O(mn), is the big O time complexity to process each of the two instances of lights and switches problems into circuit class objects. Since this is done twice for the two instances and since 2 is dropped as a constant coefficient in the big O representation of time complexity, Step 2 = O(nm).

In Step 3, each empty element the “lights” property list of each circuit class object is populated with clause representations of each light in O(nm) time because for each of the two circuit class objects:

- The fill_lights method defined in the circuit class is called to represent each light as clauses in terms of variables P and Q, each call runs in O(m) time as described above.
- For each element in the class object’s “connections” property matrix, this step indexes into the outer list of the class object’s “lights” property matrix in O(1) time and replaces all instances of a P or Q variable in the two clauses represented at the outer list index element of the “lights” property matrix. All instances of a variable are replaced with an index number in O(1) time, because there is up to some constant number of characters to search though in the representation of two clauses, and because there is up to a constant number of instances of a P or Q variable to be found in the representation of two clauses. The number of elements in the class object’s “connections” property matrix is upper bounded by nm, since the matrix’s outer list has an index for each of the n switches, and each switch’s index has a list of up to m elements. Since some constant number of operations are performed for all elements of the object’s “connections” property matrix, and the number of elements in the matrix is upper bounded by nm, operations are performed for all elements in O(nm) time.
- Lesser terms are dropped in the big O representation of time complexity, so the greatest time complexity term of the above described mechanisms, O(nm), is Step 3’s big O runtime for each of the 2 circuit class objects. Since 2 is a constant coefficient, it is also dropped in the big O representation of time complexity for both circuit class objects, so Step 3’s overall big O runtime is also O(nm); Step 3 = O(nm).

In Step 4, the 2m clauses from the “lights” property matrix of each circuit class object are added to the circuit class object’s corresponding two_sat class object. Each clause is passed as an argument to a two_sat class method in O(1) time, so the 2m clauses of each circuit problem are passed as an argument to a two_sat class method in O(m) time.

To reduce our original problem to a 2-SAT problem matching the input formatting of the blackbox, our reduction runs Step 1 in O(1), Step 2 in O(nm), Step 3 in O(mn), and Step 4 in O(m). Since lesser terms and constant coefficients are dropped in the big O representation of time complexity, all steps of our reduction run in O(nm), which is also upper bounded by polynomial time complexity.
*Note: Confirmed with John in his 3/13 office hours that O(nm) is within polynomial time.*

# Proof that 2-SAT Reduction is Satisfiable Iff there is a Way to Turn Off All the Light(s)

## Definition of Original Problem

This original switches and lights problem is a boolean satisfiability problem pertaining to whether circuits of lights and switches have some possible switch configuration such that all lights can be turned off. The original problem should return yes, satisfiable, when there is some switch configuration to turn off all lights and should return no, unsatisfiable when there is no possible switch configuration to turn off all lights.

## Preface: Proof Method

To prove that our 2-SAT reduction of the original switches and lights problem is satisfiable if and only if there exists some switch configuration that would turn off all of the lights:

- First, we will prove that our 2-CNF representation of any single light's possible switch configurations must evaluate to true if and only if the switch configuration possibility results in the single light being turned off.
- Then we will prove that joining every 2-CNF representation of each light's switch conditions with "and" logical operators must result in a 2-CNF formula that is satisfiable if and only if there exists a possible set of switch configurations such that all lights can be turned off.

## 2-CNF Representation of Any Single Light

### Switch Configuration Possibilities for Any Single Light

Since any single light is always connected to exactly two switches, and since switches always have exactly two possible states, there are always exactly four possible switch configurations that determine the on/off state of any single light. Below please find a representation of these four possible switch configurations for any single light, where "P" is one of the light's switches in its initial input state, where "~P" is the same switch in the state opposite its initial state, where "Q" is the light's other switch in its initial input state, and where "~Q"is the light's other switch in the state opposite its initial state.

- **( P, Q):** Switch P and switch Q in their initial input states.
- **(~P, Q):** Switch P in the state opposite of its initial state, Q in its initial state.
- **( P, ~Q):** Switch P in its initial state, Q in the state opposite of its initial state.
- **(~P, ~Q):** Switch P and switch Q in the states opposite their initial states.

### 2-CNF Representation of Any Light with an Initial State of "On"

If any light's initial state is "on", it will be represented by the following 2-CNF formula:

(P or Q) and (~P or ~Q)

The 2-CNF Formula "(P or Q) and (~P or ~Q)" evaluates as follows for each of any "on" light's for possible switch configurations:

- **( P, Q):** 2-CNF Formula "(P or Q) and (~P or ~Q)" evaluates to false.
- **(~P, Q):** 2-CNF Formula "(P or Q) and (~P or ~Q)" evaluates to true.
- **( P, ~Q):** 2-CNF Formula "(P or Q) and (~P or ~Q)" evaluates to true.
- **(~P, ~Q):** 2-CNF Formula "(P or Q) and (~P or ~Q)" evaluates to false.

As shown above, the 2-CNF Formula “(P or Q) and (~P or ~Q)” only evaluates to true if and only if one of the light’s connected switches remains in its initial state and the light’s other connected switch is in a state opposite its initial state.
This is correct, because if any light’s initial state is “on”, it will be only be turned “off” iff only one of it’s connected switches changes states, changing the light’s state to “off”.
No contradictions are possible, because if neither of the light’s connected switches change states then the light remains “on”, and because if both of a light’s connected switches change states then the light’s state is changed twice - to “off” by one switch and then back to “on” by the other switch.

Please note that this section of the proof proves only that the two clauses representing any single “on” light which must only evaluate to true iff a given set of switch states is such that the light will be turned off.
Please see the later section titled “2-CNF Formula Representing All Lights” for proof pertaining to the 2-CNF representation of an entire lights and switches problem.

**2-CNF Representation of Any Light with an Initial State of “Off”**
If any light’s initial state is “off”, it will be represented by the following 2-CNF formula:

(~P or Q) and (P or ~Q)

The 2-CNF Formula “(~P or Q) and (P or ~Q)” evaluates as follows for each of any “off” light’s for possible switch configurations:

- **( P, Q):** 2-CNF Formula “(~P or Q) and (P or ~Q)” evaluates to true.
- **(~P, Q):** 2-CNF Formula “(~P or Q) and (P or ~Q)” evaluates to false.
- **( P, ~Q):** 2-CNF Formula “(~P or Q) and (P or ~Q)” evaluates to false.
- **(~P, ~Q):** 2-CNF Formula “(~P or Q) and (P or ~Q)” evaluates to true.

As shown above, the 2-CNF Formula “(P or Q) and (~P or ~Q)” only evaluates to true if and only if either both of the light’s connected switches are left in their initial states or if both of the light’s connected switches are in a state opposite their initial state.
This is correct, because if any light’s initial state is “off”, it will be only be turned “off” iff either neither of the light’s connected switches change states, leaving the light “off”, or if both of the light’s connected switches change states, changing the light’s state twice - to “on” by one switch and then back to “off” by the other switch.
No contradictions are possible, because if only one of its connected switches changes states then the light’s state would be changed to “on”.

Please note that this section of the proof proves only that the two clauses representing any single “off” light must only evaluate to true iff a given set of switch states is such that the light will be turned off.
Please see the later section titled “2-CNF Formula Representing All Lights” for proof pertaining to the 2-CNF representation of an entire lights and switches problem.

## 2-CNF Formula Representing All Lights

### Format of Resulting Formula

By definition in the assignment description, our 2-CNF formula is conjunctions of clauses, where each clause is a disjunction of exactly two literals. In other words, each clause is two literals joined by an “or” logical operator, and all clauses are joined by “and” logical operators.

The 2-CNF representations of each light in terms of its connected switches are a conjunction of two clauses, where each clause is a disjunction of literals, and where each literal represents the state of a connected switch.
The formula representing all lights is produced by joining all clauses of all lights with “and” logical operators. Since every clause is a disjunction of literals and since every clause is joined by an “and” logical operator, this formula representing all lights meets the definition of a 2-CNF formula.

### 2-CNF Formula Satisfiable If and Only If it is Possible to Turn off All Lights

It is possible to turn off all lights when there is some set of switch states such that all lights will be off. If it is not possible to turn off all lights, there must be some contradiction between the switch states required for one light to be turned off and the switch states required for another light to be turned off, such that it is not possible for both lights to be turned off with the same set of switch states.

As previously established, the 2-CNF representation of any single light is a conjunction of two clauses, which will only evaluate to true if the given set of switch states is such that the light will be turned off.

In the 2-CNF formula representing all lights, the 2-CNF representations of each single light are joined by “and” logical operators. By the definition of the “and” logical operator, the entire formula of conjunctions of clauses will only evaluate to true if every clause returns true, and will otherwise evaluate to false if one or more clauses evaluate to false with the given literal input values. In the context of our switches and lights problem reduction, this means that the 2-CNF formula representing all lights would only evaluate to true, if and only if a given set of switch states is such that all lights will be off.

By definition, a 2-CNF formula is satisfiable if and only if there exists some set of inputs such that the formula evaluates to true.

So our 2-CNF formula representing all lights is satisfiable if and only if there exists some possible set of switch states is such that all lights will be off, since a 2-CNF formula is satisfiable if and only if there exists some set of possible inputs such that the formula evaluates to true, and since our 2-CNF formula representing all lights would only evaluate to true if and only if a given set of switch states is such that all lights will be off.