ICS2210
Data Structures and Algorithms 2
Documentation

Daniel James Sumler, 0107297A Bsc. Computing Science Yr2

Table of Contents

Introduction

In this assignment, I was tasked with developing a DPLL algorithm as well as a Huffman Coding Tree algorithm. Both algorithms and their corresponding helper methods were coded in **Java**.

I approached these tasks by first understanding the problems at hand and conducting research online and in the supplied class notes. I believe it is important to fully understand any algorithms which I am tasked to implement, in order to make them as efficient as possible.

Both algorithms were coded with not only memory efficiency in mind, but also user interface in order to make it as readable and useable as possible. This is done by using helper functions in ‘Part 1’ by printing out the clauses after each rule is used, and having clear error handling in both parts of the assignment in order to guarantee that the program has a soft exit. Each error is clearly indicated with its own unique error message.

Both implementations have their flaws, and these are clearly stated in the Evaluation section of each task.

Part 1 – Boolean Satisfiability

Approaching the Boolean Satisfiability Problem

The Boolean satisfiability problem is defined as the problem of determining whether there exists a suitable interpretation that satisfies (SAT) a given formula.

A simple example of this would be:

For an expression **(x)(x,!y,!z)**, if **x = True** the expression is satisfied.

Else, if there is no possible interpretation which makes the expression satisfiable, the expression is considered unsatisfiable (UNSAT).

An example of this would be:

For an expression **(x)(!x)**, if **x = True** the expression is unsatisfied.

For an expression **(x)(!x)**, if **x = False** the expression is unsatisfied.

All possibilities have been explored, and therefore the expression remains UNSAT.

In the DPLL algorithm, these possibilities are narrowed down by using techniques such as the **1 Literal Rule**, the **Pure Literal Rule** and the **Splitting Rule**. These rules work together to make the algorithm more efficient, as guessing every possibility would greatly increase its time complexity.

The approach I took to code the DPLL algorithm used to test all the interpretations of each expression as one in which I store each clause as a String in an *ArrayList* of Strings. With this approach, the clauses can be easily manipulated and can include any number of literals, to avoid being restricted to having 2-SAT or 3-SAT expressions.

How the program functions

In the first part of my coded solution, the program first checks whether the inputted statement is syntactically correct. In order to do this, the statement is thought of as a Deterministic Final State Automata. This is done to prevent the program from crashing if an incorrect input is given, and to make it easier to add any new syntactical features if the need arises. Below, one can find a diagram of the DFSA as well as the code used to simulate this.

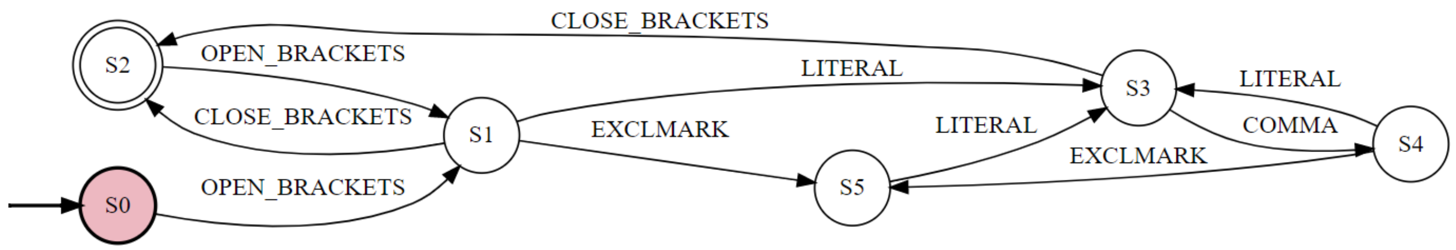


Figure 1 - DFSA of lexer - made on http://ivanzuzak.info/noam/webapps/fsm_simulator/

In the above diagram, one can see that the DFSA is designed to make sure the program never crashes when entering an input statement. The tokens used in the transitioning between states, as well as the states themselves can be found in *Part1Tokens* and *Part1States* respectively.

```
//Rows of states
static Part1States[] Rows = {Part1States.S0, Part1States.S1, Part1States.S2,
Part1States.S3, Part1States.S4, Part1States.S5, Part1States.SE};
//Columns of tokens
static Part1Tokens[] Columns = {Part1Tokens.OPEN_BRACKETS, Part1Tokens.CLOSE_BRACKETS,
Part1Tokens.LITERAL, Part1Tokens.COMMA, Part1Tokens.EXCLMARK, Part1Tokens.INVALID};
//Transition table
static Part1States[][] symbolTable =
        //OPEN_BRACKETS      CLOSE_BRACKETS    LITERAL      COMMA          EXCLMARK
/*S0*/ { {Part1States.S1, Part1States.SE, Part1States.SE, Part1States.SE, Part1States.SE,
INVALID
Part1States.SE},
/*S1*/    {Part1States.SE, Part1States.S2, Part1States.S3, Part1States.SE, Part1States.S5,
Part1States.SE},
/*S2*/    {Part1States.S1, Part1States.SE, Part1States.SE, Part1States.SE, Part1States.SE,
Part1States.SE},
/*S3*/    {Part1States.SE, Part1States.S2, Part1States.SE, Part1States.S4, Part1States.SE,
Part1States.SE},
/*S4*/    {Part1States.SE, Part1States.SE, Part1States.S3, Part1States.SE, Part1States.S5,
Part1States.SE},
/*S5*/    {Part1States.SE, Part1States.SE, Part1States.S3, Part1States.SE, Part1States.SE,
Part1States.SE},
/*SE*/    {Part1States.SE, Part1States.SE, Part1States.SE, Part1States.SE, Part1States.SE,
Part1States.SE}
```

In the above snippet of code, one can see how the symbol table is made. Once a token is identified from the user's input, the program checks the symbol table to check which state it should transition to next.

After this is done, the program separates the user's clauses into separate Strings inside an *ArrayList*. The choice to use an *ArrayList* was made in order to make it easier to delete clauses and make the whole process more reliable.

Next, the *ArrayList* of clauses is passed into the *DPLL* function. Firstly, this finds any clauses which would make the whole statement unsatisfiable (i.e. **(p)(!p)**). If an example of this is found, false is returned. Else, the program carries on running.

The next step is to find and remove any trivially satisfiable clauses (i.e. **(p,!p)**). These are found and removed from the statement.

Next, the 1 literal rule is applied to the statement of clauses. This works by determining whether there exists a string of length smaller than or equal to 2 in the *ArrayList* and calls the *Apply1LiteralRule()* function using the found literal. This function removes the negation of the literal from all clauses and removes clauses containing the literal itself. This is done using the helper function *CheckForNegation()* to alter the strings. The *ArrayList* is then checked to see whether it contains an empty clause or an empty list, causing the method to return false or true respectively.

After this, the program calls the *ApplyPureLiteralRule()* function in order to scan each string and determine whether there exists a literal which remains positive throughout every clause. If one is found, every clause containing said literal is removed from the list, and this is repeated for every literal (**w**, **x**, **y** and **z**).

Once these steps have been completed and there is not yet a return value, the program chooses a suitable literal in order to use the *Splitting Rule*. In this case, the program chooses the most common literal, using *HashMaps*, to use the splitting method on. This is done in order to reduce the chance of falling into an infinite loop, although this may still be possible. A new *ArrayList* is created with the added element, and the *DPLL* method is called again until a return value can be determined.

Testing the program

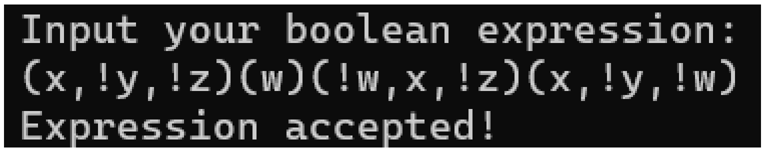
In this section of the documentation, I will be testing every **main function** in the Part1 class. An input will be given to the program, and it the output will be matched with a pre-determined expected output value.

Testing expression parsing

In order to test whether the program's syntactic analysis works, I will enter a syntactically correct statement and an incorrect one. The program should output whether the expression has been accepted or not.

Correct Statement:

The expression **(x,!y,!z)(w)(!w,x,!z)(x,!y,!w)** was entered into the program:

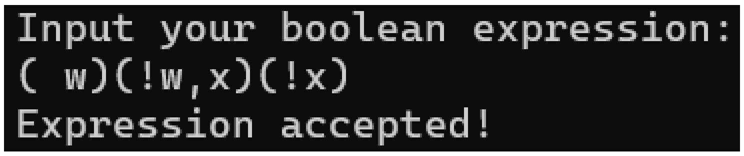


```
Input your boolean expression:
(x,!y,!z)(w)(!w,x,!z)(x,!y,!w)
Expression accepted!
```

The actual output matched the expected output, therefore the test **passed**.

Correct Statement with whitespace:

The expression **(w)(!w,x)(!x)** was entered into the program:



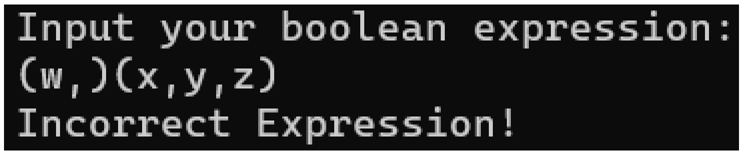
```
Input your boolean expression:
( w)(!w,x)(!x)
Expression accepted!
```

The actual output matched the expected output, therefore the test **passed**.

Incorrect Statement:

Expected output: The program rejects the statement and shows a clear error message.

The incorrect expression **(w,)(x,y,z)** was entered:



```
Input your boolean expression:
(w,)(x,y,z)
Incorrect Expression!
```

The actual output matched the expected output, therefore the test **passed**.

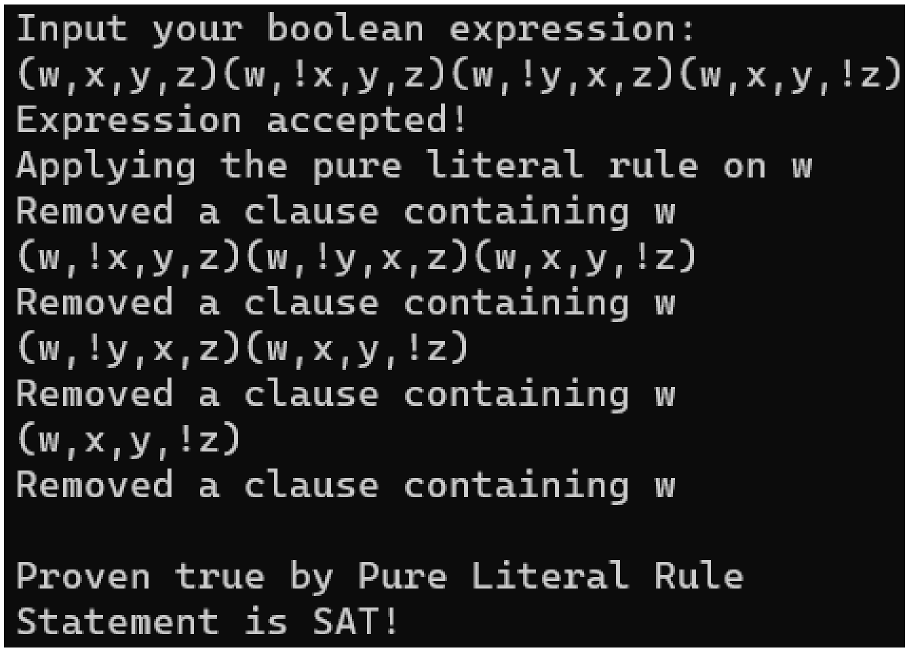
Testing program functions

Testing the Pure Literal Rule:

Testing that an expression that can be declared satisfiable solely by use of the *Pure Literal Rule* is solved correctly.

Input: (w,x,y,z)(w,!x,y,z)(w,!y,x,z)(w,x,y,!z)

Expected output: The statement is declared SAT and the program shows every step it used in order to reach this conclusion.



```
Input your boolean expression:
(w,x,y,z)(w,!x,y,z)(w,!y,x,z)(w,x,y,!z)
Expression accepted!
Applying the pure literal rule on w
Removed a clause containing w
(w,!x,y,z)(w,!y,x,z)(w,x,y,!z)
Removed a clause containing w
(w,!y,x,z)(w,x,y,!z)
Removed a clause containing w
(w,x,y,!z)
Removed a clause containing w

Proven true by Pure Literal Rule
Statement is SAT!
```

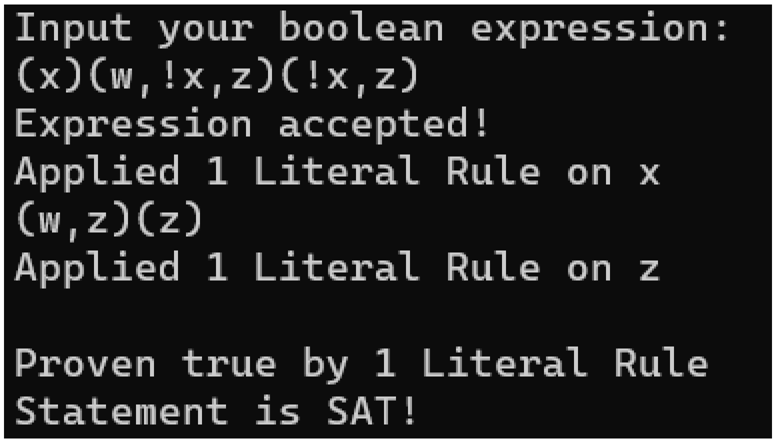
The actual output matched the expected output, therefore the test **passed**.

Testing the 1 Literal Rule with a positive unit clause:

Testing using an expression that can be proved SAT, solely by using the *1 Literal Rule*.

Input: (x)(w,!x,z)(!x,z)

Expected output: The expression is declared SAT and the program shows every step it used in order to reach this conclusion. Every instance of ***!x*** should be removed from each clause and clauses containing *x* should be removed completely.



```
Input your boolean expression:
(x)(w,!x,z)(!x,z)
Expression accepted!
Applied 1 Literal Rule on x
(w,z)(z)
Applied 1 Literal Rule on z

Proven true by 1 Literal Rule
Statement is SAT!
```

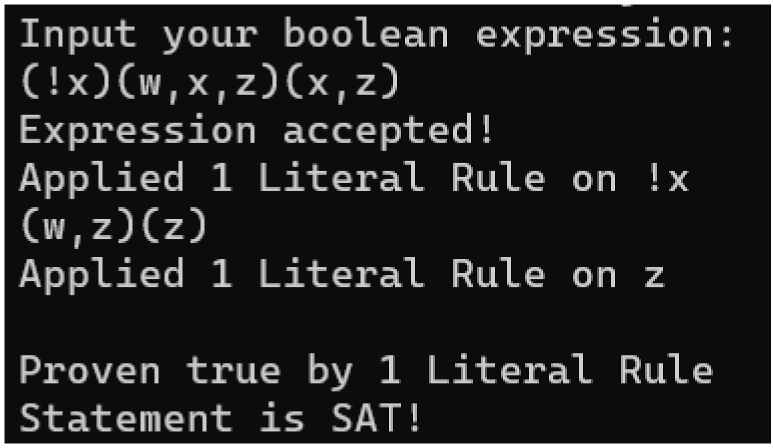
The actual output matched the expected output, therefore the test **passed**.

Testing the 1 Literal Rule with a negative unit clause:

Testing using an expression that can be proved SAT, solely by using the *1 Literal Rule*, using a negative unit clause.

Input: (!x)(w,x,z)(x,z)

Expected output: The expression is declared SAT and the program shows every step it used in order to reach this conclusion. Every instance of **x** should be removed from each clause and clauses containing **!x** should be removed completely.



```
Input your boolean expression:
(!x)(w,x,z)(x,z)
Expression accepted!
Applied 1 Literal Rule on !x
(w,z)(z)
Applied 1 Literal Rule on z

Proven true by 1 Literal Rule
Statement is SAT!
```

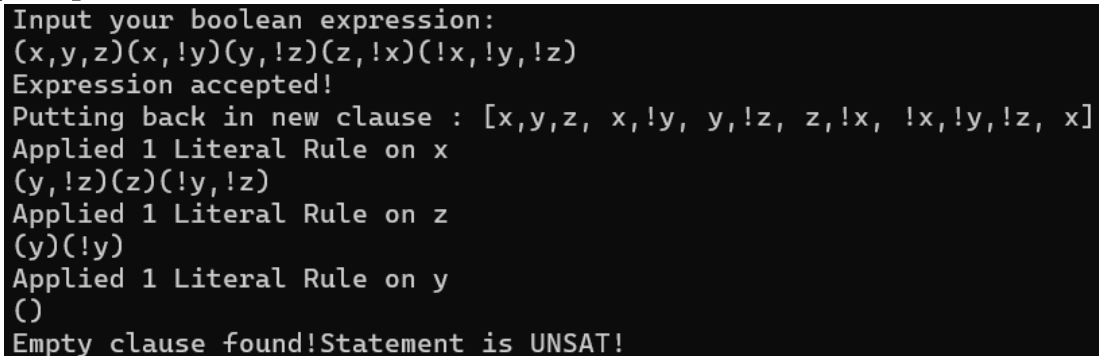
The actual output matched the expected output, therefore the test **passed**.

Testing the Splitting Rule:

Testing using an expression that is solved by using the *Splitting Rule* in conjunction with other *DPLL* rules.

Input: (x,y,z)(x,!y)(y,!z)(z,!x)(!x,!y,!z)

Expected output: The expression is declared SAT and the program shows every step it used in order to reach this conclusion.



```
Input your boolean expression:
(x,y,z)(x,!y)(y,!z)(z,!x)(!x,!y,!z)
Expression accepted!
Putting back in new clause : [x,y,z, x,!y, y,!z, z,!x, !x,!y,!z, x]
Applied 1 Literal Rule on x
(y,!z)(z)(!y,!z)
Applied 1 Literal Rule on z
(y)(!y)
Applied 1 Literal Rule on y
()
Empty clause found!Statement is UNSAT!
```

The actual output matched the expected output, therefore the test **passed**.

Practical Applications of SAT

As one could guess, there are many practical applications in which Boolean Satisfiability can be used. This method is used due to the simple fact that it is NP-complete, meaning that it can be solved in polynomial time.

Boolean Satisfiability is used widely in the industry. For example, products assembled out of standardized components can make use of this. This includes computers, cars, and various others.

Other, more electronic applications of SAT include model-checking of finite state systems, debugging programs, and software testing to name a few.

Another, more obvious practical application would be in the realm of hardware verification. This comes in handy when verifying whether logic gate expressions are satisfiable, as when used in the industry, can become very complicated expressions.

It is very important to use Boolean Satisfiability in the areas mentioned above, as it is a vital part of testing. If there comes a point in which there exist no solutions to an expression, then this could be a devastating flaw in the system.

As one can clearly see, Boolean Satisfiability is important in a lot of electronic sectors of manufacture. Thanks to the efficient DPLL algorithm, Boolean Satisfiability is now even more common to find being used, due to the short amount of time it takes to prove or disprove an expression.

Source: https://eprints.soton.ac.uk/265340/1/jpms-wodes08.pdf, Joao Marques-Silva

Evaluation

Overall, I believe that my work done for this task was done adequately and met all requirements that were asked for. As stated before, efficiency, reusability of code and user-friendliness were all taken into account in order to make a quality solution. If I were to critique my work, it is possible that a different method could have been used instead of String manipulation, but this was chosen in order to work together with my parser.

Known Flaws

A known flaw is that the program may fall into infinite loops, due to the way the literal is chosen before commencing the *Splitting Rule*. Lots of different methods were tried and tested during development, and selecting the most common literal seemed to be the most reliable out of all of them. A possible temporary fix for this problem would be to go to **line 335** of the **Part1.java** and use either the left Clause or the right Clause instead of both at once.

All in all, my program is an accurate and reliable way to determine whether an expression is SAT or UNSAT.

Part 2 – Data Compression

Approaching the Huffman Coding Method

The Huffman Coding method is one which compresses data, irrespective of data type. For example, an image, audio file or document could be sent from one station to another, and the same method would be used.

This method works by looking at the data stream that makes up the file that is to be sent and takes note of frequently occurring characters. Each character is taken, together with its frequency and is sorted into a **Huffman Tree**, in which the branches represent the sum of the frequencies of its children nodes, while the leaves contain a character.

Once the tree has been formed, the **Huffman Code** can be found. This is determined by traversing the tree until a leaf node is encountered. For every left branch traversed, a **0** is added to the code and a **1** for every right branch traversed.

Once a leaf is reached, the code is recorded and attached to the character in that leaf. An example of a Huffman Tree and how its code is derived can be seen below.

Data compressed: **aabccc**

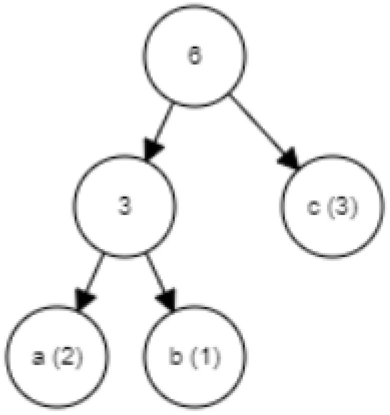


Figure 2 - A Huffman Tree - made on https://people.ok.ubc.ca/ylucet/DS/Huffman.html

By traversing the tree we can see that:

a = 00, b = 01, c = 1 which means that **aabccc = 000001111**

In contrast with normal ASCII code, we can see a huge difference:

aabccc = 01100001 01100001 01100010 01100011 01100011 01100011 00001010

In order to code this algorithm, I utilized *HashMaps* and a custom *Node* class in order to match each character with its frequency. *PriorityQueues* were also used in order to keep the list ordered, as per the Huffman Code requirements. For a full, detailed explanation on my implementation, see below.

How the program functions

As soon as the program is run, the user is prompted to enter a filename. If an incorrect filename is entered, the program softly exits and presents a valid error message to the user.

If the file is successfully found, each character in each line is individually checked in order to match the regular expression *[a-zA-Z0-9]** which only allows the file to contain letters and numbers. If a different symbol is found, the program softly exits and presents the user with a valid error message. Once the input is deemed correct, each line is placed in a different index in an *ArrayList.*

Next, the *Huffman()* function is called. This declares a *HashMap* and a *PriorityQueue*. The *HashMap* is first used in order to record each unique character and how many times they appear in the file. Next, each record is turned into an instance of the *Node* custom class. Each of these *Nodes* is added to the *PriorityQueue* where they are sorted based on their frequency, using the pre-defined *Comparator*, *NodeFrequency*.

```
private static Comparator<Node> NodeFrequency = (n1, n2) -> {
    return n1.getFrequency() - n2.getFrequency();
};
```

The next step is to combine the frequencies of two characters into one frequency node, as per the *Huffman Code* method. This is done by polling the

PriorityQueue and extracting each *Node* and using them to create a new parent *Node*, in which the original nodes are used as children and the new *Node* is a summation of its children's frequencies. The code for this can be seen below:

```
while (NodeQueue.size() != 1) {
    Node l = NodeQueue.poll();
    Node r = NodeQueue.poll();

    Node n;
    if(r != null) {
        n = new Node(l.getFrequency() + r.getFrequency(), l, r);
    } else{
        n = new Node(l.getFrequency(), l, null);
    }

    NodeQueue.add(n);
}
```

The next step taken by the program is the generating of the *Huffman Code* by using the *Huffman Tree*. This is done recursively by the *GenerateCode* function. A root *Node* is supplied and the program repeatedly accesses the left and right children of the given node. For every left child accessed, a **0** is added to the *Huffman Code*, and a **1** for every right child. This runs for every branch until a leaf node is encountered on every branch. The code for this can be seen below:

```
private static void GenerateCode(Node n, String code, HashMap<Character, String>
HuffmanTree) {
    if (n != null) {

        if (n.getLeftChild() == null && n.getRightChild() == null) {
            HuffmanTree.put(n.getChar(), code);
        }

        GenerateCode(n.getLeftChild(), code + "0", HuffmanTree);
        GenerateCode(n.getRightChild(), code + "1", HuffmanTree);

    }
```

The last step the program takes before outputting the *Huffman Codes* to the user is organizing the codes in ascending order based on their ASCII values. This is done by converting each code to decimal and placing them in order in a *PriorityQueue*. These values are then put in a *TreeMap*, together with its corresponding character. These values are then matched with the original

HashMap and each character is outputted to the user with its corresponding code, in ascending order.

Testing the program

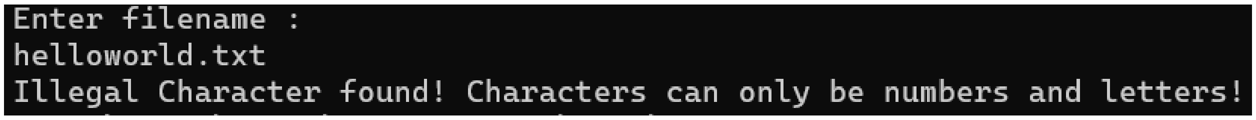
In this section of the documentation, I will be testing every **main function** in the Part2 class. An input will be given to the program, and it the output will be matched with a pre-determined expected output value.

Testing File Reading

File containing illegal characters:

Input: A text file containing the characters “abcd$efg123” will be created and used by the program.

Expected Output: The program rejects the file and gracefully exits.



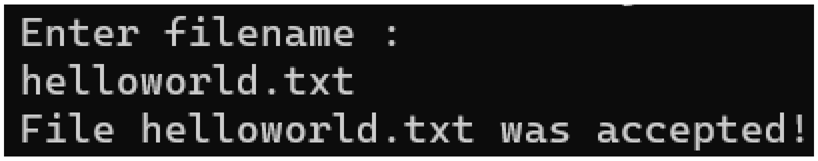
```
Enter filename :
helloworld.txt
Illegal Character found! Characters can only be numbers and letters!
```

The actual output matched the expected output, therefore the test **passed**.

File containing legal characters:

Input: A text file containing the characters “HelloKristian” will be created and used by the program.

Expected Output: The program accepts the file and continues operating.



```
Enter filename :
helloworld.txt
File helloworld.txt was accepted!
```

The actual output matched the expected output, therefore the test **passed**.

Testing the Huffman Code:

Testing the Huffman Code is correct using letters:

Input: A text file containing the characters "aaabbcaddddd" will be created and used by the program.

Expected Output: The program accepts the file and outputs the following Huffman Codes :

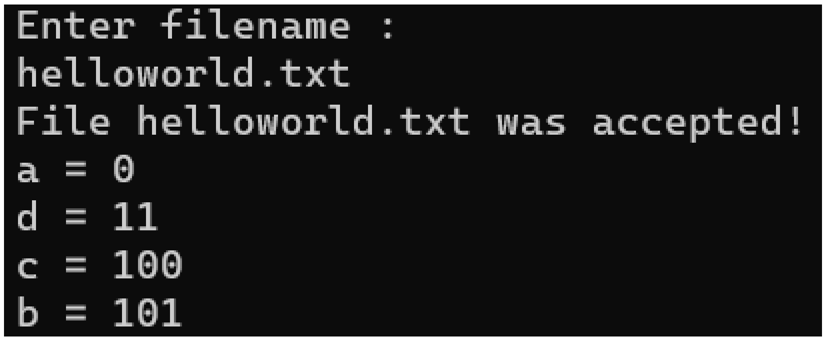
a - 0

b - 010

c - 100

d - 11

The above codes were worked out by hand.



```
Enter filename :
helloworld.txt
File helloworld.txt was accepted!
a = 0
d = 11
c = 100
b = 101
```

The actual output matched the expected output, therefore the test **passed**.

Testing the Huffman Code is correct using numbers:

Input: A text file containing the characters “000112222999” will be created and used by the program.

Expected Output: The program accepts the file and outputs the following Huffman Codes :

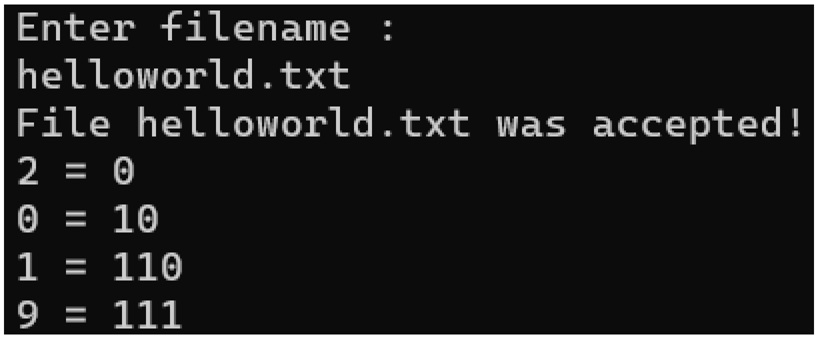
‘2’ - 0

‘0’ - 10

‘1’ - 110

‘9’ - 111

The above codes were worked out by hand.



```
Enter filename :
helloworld.txt
File helloworld.txt was accepted!
2 = 0
0 = 10
1 = 110
9 = 111
```

The actual output matched the expected output, therefore the test **passed**.

Evaluation

I believe that I have created a solution to the assignment task that is not only efficient and user friendly, but also very reusable, as one could very easily add another literal and symbols to the accepted syntax if they wish.

On top of this, correct error messages are shown when a problem is encountered, and a way to get the program to crash is yet to be discovered.

Each time a test was performed with the program, a hand-written test was also taken out, and the program matched the required values every time.

Known Flaws

A known flaw of this solution, is that sometimes the *Huffman Tree* might be unbalanced, on either the left or right. Many solutions have been tried to solve this issue, but none have been successful.

Overall, in my opinion, this is a solution which meets all the criteria set out before me.

Please note that this solution does not accept text files containing whitespaces. If you wish to use whitespaces, please go to Part2.java and uncomment line 32.

The code and ideas for this solution were loosely based on a solution by:

Various authors (Date unknown) Huffman Code | Greedy-Algo 3, https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/,

Conclusion

To conclude, I believe that both tasks were executed to the best of my ability, with efficient and re-usable code and intuitive design.

Although some flaws were identified in the code, I still stand by the fact that they are reliable enough to do their tasks properly. As stated before, numerous attempts were made to try and fix said issues, but none were found which guaranteed a 100% success rate.

During this assignment, I learnt the importance of Boolean Satisfiability and why algorithms such as the DPLL algorithm are so important in the industry. Not only this, but I also learnt how important and necessary data compression is, as the difference between compressed and uncompressed data is very noticeable.

Statement of completion – MUST be included in your report

Item	**Completed (Yes/No/Partial)**
Part 1 – Accepted and parsed input.	Yes
Part 1 – Implemented DPLL	Yes
Part 2 – Implemented Huffman coding	Yes
If partial, explain what has been done	

Marking Breakdown

Description	**Marks allocated**
Parsing CNF expressions	10%
Implemented DPLL	30%
Implemented Huffman coding	15%
Section about practical applications of SAT	10%
Evaluation and testing of artifacts	15%
Overall report quality	20%