Fontys
University of Applied Sciences

ALE 1
DESIGN DOCUMENTATION

FHICT English Stream

Abstract

This documentation describes the design of an expression analyzer application. The input must contain a logical proposition and the output is represented by a dnf form, nand form. Truth tables are also provided along with a binary tree.

Maximilian Dobre
341887

Introduction

This is the design document for an expression analyzer application. The application presents the following functionalities: parsing a propositional expression and building a binary tree, computing truth table and hash code, creating a simplified truth table, both disjunctive normal form and simplified disjunctive normal form, and computing a nandified form based on these results. The application is written in python and the chosen testing framework is pytest. Making of the application proved to be a challenge at times and extensive amount of time was dedicated to debugging the code; this was especially the case of simplifying the table functionality.

As part of this document you will find all the implemented functionalities discussed, as well as a description of the user interface and the implemented tests.

Assignment 1: Parse + Tree

First I tried to parse the tree by calling a recursive function over the input expression. The idea was that a left operand and a right operand would be built and sent as parameters in the next recursive call. I realized that this approach involves a lot of checking on the input, that the code resulted from this is very complex and that is proned to many errors. I think that the main issue with the first approach was that building the tree was done in a BFS-like order, building a level before moving to the next, while the second one doesn't require checking because it builds the tree according to each element the algorithm sees, and going as deep in the tree as it was required (as indicated by the open brackets/ commas).

The function required an additional counter parameter which would have kept track of the of the number of nodes created and would serve as a unique identified for each node. The counter was required as each node of a graphviz tree had to have a unique name.
Rendering the tree using graphviz was rather a formality, all I had to do is traverse the parsed binary tree, construct an additional tree data structure that is used by the graphviz API and perform a rendering call to that API. For clarity of the code, this was done by a main rendering function that would first call a recursive function to create the graphviz tree.

Assignment 2: Truth table + Hash code

Populating the truth table was done per column by filling alternating groups of 0's and 1's of size 2^(n-i) where n is the total number of columns and i is the column index of the predicate/column, ranging from 1 to n. After the table was generated in the form of a matrix, each column was assigned to the corresponding predicate. The chosen data structure for this is a dictionary.
Computing the hash code involved computing the result value of the expression for each particular instance (or row in the table) in the total number of 2^n

number of possibilities. To obtain the hash code all there is to do is display the list containing the result values in reverse order.

Computing each expression result value was done with the help of the parsed binary tree. The tree had its leaves first initialized with the corresponding values according to the truth table. This was done for each row in the table. Than, for each node in the tree its value was calculated with a function that would take into account the allowed operators: '&, |, >, =, ~'.

I think that the only more challenging part of assignment was populating the truth table as I described before. I believe an easier approach would have been to generate all numbers between 0 and 2^n – 1, convert them to binary and use each digit to fill the table cells. This would have been held in a matrix as well, the only difference being that in this case populating would have been done on a per-row basis so the matrix would have had to be looped through again to append the values in the dictionary, even if the values could have been added on the fly too. I think the biggest advantage of the former method is raising awareness about how a table can be populated and how it can be easier checked if the table is constructed correctly or not.

Maybe an important implementation note should be made here: the truth table and the expression result values were stored in 2 different data structures for easy use. It gave a finer-grained control on what I could do; not having to send the whole truth table as parameter in cases where I wanted only the expression result values array.

A different way to store the truth table would have been by storing the columns inside the corresponding leaf nodes. Or creating an additional data structure that would hold both the name of the predicate and the list of its possible values, a data structure that could be indexed, very similar to a list.

Assignment 3: Simplify

This was definitely the most difficult part to implement. A simple look to the code will show that. It was because of my try of not iterating too many times over some data structures that led to the code being so nested. I did try to think of a way compute a simplified table that would use some sort of property that would work around having a lot of computations, but I did not manage to find any. The idea of my implementation is quite straightforward though: basically, for every value in the truth table I select the rows which have a different value on the column of the value I currently am at. I compare the current value row with each of the candidate rows and see if there is at least one common element column-wise. This indicates that the 2 rows may lead to a simplified row (given that the simplified row does not expand to a row which has an expression result value of 0). If the 2 rows can be simplified, I build a simplified row and check if it indeed is valid (as in, all the initial table's rows that the simplified row stands for have an expression result value of 1). Then I 'mark' the 2 rows as simplified by appending their index into a list and checking every row against it. I based this on the observation that simplification is commutative (as in, if you simplify A with B and the result with C, it will yield the same result as simplifying A with C and the

result with B) so there is no need to compare rows that have been already simplified. Another approach would be to generate all the possible simplifications, even if you consider the same rows twice and compute a simplification in both ways (A with and B with A), and then remove the duplicate rows from the intermediate simplified table before checking for the possibility of simplification.

One of the sidecases of this part of the algorithm is the case of odd number of rows that can be simplified. As my approach is to take ‘adjacent’ rows and obtain a simplification out of them, it is possible to have the odd row out of the process. I tackled this case by appending the last row to the intermediate simplification table. This, however, did not solve the case where there is only one row with an expression result value of 1 so the algorithm cannot get in the situation to see ahead and append the row. Therefore, a case distinction has been made.

Checking whether a table can be simplified or not involved the condition that there are 2 different values on any column of the table, and that the rows containing those values have at least a non-star common value situated on the same row.

A very important note here is the fact that the simplified functionality in my application is designed to not give row full of stars for a tautologic expression. In the explanation of my algorithm I mentioned that after I computed a full intermediate table I check for the existence of a common non-star element on any 2 rows that have different elements on at least one same column. The key word here is ‘non- star’. This is enough to prevent a ‘tautologic row’ from appearing as final result of the simplification, since a ‘tautologic row’ cannot appear during a simplification (as in, it cannot appear along other rows in the simplification table), given the way the table is constructed. Thus, the algorithm stops one step before the ‘tautologic row’ and makes any expression depend on at least one variable/predicate. ‘Tautologic row’ stands for a row that only contains stars.

This is maybe the only place in this application that the input to a function is partially trusted. It goes on the assumption that the input is a truth table that can’t generate ‘tautologic rows’ between checks, that is before the intermediate simplified table to be fully computed. However, this is independent on the user input, as he/she does not have control on how the truth table is computed, but only on the input expression.

A sidecase I encountered is when the input has only one predicate, and his rows can be simplified. I added a check on the number of predicates so that the simplified table stays exactly as the original one.
Another sidecase I found is when the input is provided in such a way that it leads to a last intermediate table (the one before the output one) that consists of 3 rows with the first 2 generating a tautologic row. No other simplifications are found with the last row so they are ignored according to the checks. The last row is appended to the simplified truth table as I mentioned before, leading to a truth table that is missing the first 2 rows. I solved this by adding a check on whether the obtained simplified row is tautologic or not, keeping track of all the rows that lead to such case and verifying at the end if they have been simplified with any other row. If that is not the case, they are appended to the simplified table as well.

I believe that this assignment needs some more attention compared to the other ones, so I will describe the functions I created and also the parameters:

compute_simplified_predicates: this is the main simplification function, it calls all the others and has 4 parameters: the initial truth table and the expression result values for its rows, an intermediate simplified truth table and its expression result values, which contain only 1's since the rows are already coming from rows that have 1 as expression result value. The purpose of the first 2 parameters are sent for computation to another function, while the purpose of the last 2 parameters is obvious, as this function is a recursive one.

compute_intermediate_simplified: computes intermediate simplified table, it is where the actual computations are made; it has the same 4 parameters as the main simplification function. The only difference is that in this case there was made a call that checked a simplification against the truth table along with its expression result values and the current table to be simplified along with its result values have been sent for further computation. The result of the computations are stored in some additional data structures as data structure that is processed cannot be modified in the same time.

generate_expansion: takes the simplified row to be checked, the initial truth table and its expression result values. It does exactly what the name suggests: it generates all the rows the simplified row stands for and checks whether one of the expression result values of those rows is 0. It returns a boolean value as a result of this check. The function is recursive and its stopping condition is when there are no other 'expansions' that can be generated from the current row and checks for the expression value of that row.

append2dict: appends the given row to a dictionary. It is essential that the dictionary keys are ordered before appending is done, otherwise it would lead to having the values set to the wrong predicates. This function is called after it is confirmed that the simplified row is valid, and it is added to a temporary data structure.

check_table_simplified: checks whether the truth table given can be simplified. The condition is that there is at least a pair of different values belonging to the same column, and that the rows of those values have at least a non-star common value. This function provides the stopping condition for the whole simplification process.

Initially, all these functions were part of a single one. But as the algorithm proved to not be working as expected, it has been split up in the ones I already described. Additional checks had to be implemented compared to my first version, therefore checking if a row has n-1 stars and comparing 2 rows and helper for adding list to dictionary were implemented in different functions.

Assignment 4: Normalize

Computing the disjunctive normal form (dnf) was done by a mix of directly building the string and using the Node class.

My approach was that first a line dnf is created (this involved having all the predicates, negated or not, connected by the '&' operator) and connecting all of the line dnf's with '|' operator.

The line dnf was computed by creating a list of nodes that represented each of the predicates or their negation parents, in case of a node that had to be negated because of its false value in the corresponding row of the truth table. This is done by a function that returns a string that represents the dnf for that row. The function takes out the element that is processed and calls recursively on the rest of the list until only one element is in the list and it is returned.
The next step was to gather all of the line dnf's in a matrix and follow a similar procedure to the line dnf function. This time, the line dnf's were taken out of the list and a recursive call on the rest of the list is made until all the elements are processed. The function returns the final dnf of the expression.
From a design standpoint, the functionality is split into 3 functions: a main one that creates the node list and calls both of the other 2 functions. One that builds a line dnf and one the builds the full dnf based on the line dnf's.

Computing the simplified truth table was almost identical, except that it does not require an expression result values list since it is known all the rows have true value in a simplified table, and the truth table may hold a third value besides the binary ones, the '*'.

Another way this could have been done was to send a list of strings instead of nodes in order to compute the line dnf, but the algorithm would have behaved the same. Creating a binary tree to generate the dnf just seemed to add a big overhead for otherwise simple operation.

An important thing to note here, and can be seen in tests as well, is that for an input that contains only one predicate with a single value of 0, the dnf computer will not output its negated variant, but the predicate itself. This case does not take place though, unless there is wrong input from within the application, a truth table with one predicate must have exactly 2 values.

Assignment 5: Nandify

This assignment posed problems bigger than expected.
The core idea of nandifying an expression is based on the fact that, starting from an already existing dnf, the nandification can be computed by negating the dnf twice, and eliminate one of the negations by turning every 'or' operator between the line dnf's into an 'and' and negate every line dnf. This has the huge advantage of narrowing down on the number of possibilities an input can take down to the form the dnf generally has.

The real challenge though, was writing the whole expression in ASCII nand format by using the '%' operator. Since the operator has to be binary, that means every negated line dnf would be on the first position in the operand order, and wrap all the other negated line dnf's in a '%' operator. The issue is that such a wrapping adds an unwanted extra negation to whatever it contains, inverting the natural truth value as well as the operators (from 'and' to 'or' and vice-versa). Therefore, each time a '%' operator was encountered, it had to be negated by wrapping it in

another nand that has as both children the initial nand. This was also done for each line nand, as the same issue would have been present there as well.

To achieve this goal easier, I used a binary tree that would hold a nandified expression for every line as well as the final table. Building the tree was not a very easy task, but it didn't pose too big problems either; all I had to do is compute a temporary line tree that would represent the nand for a line of the table, similar to computing the dnf.

I did have a few different attempts in how I should tackle this functionality. First, I tried doing the computation by using plain strings instead of binary trees, but it proved to be too of a difficult task since I had to negate every '%' except the ones that were already representing a negation (the negation of a predicate, according to its value in the truth table). After I decided on using binary trees, an approach would have been to generate an 'unfixed' binary tree first that would need to have all the nands negated (with the only exception that I mentioned above). I realized that the code is much shorter and involves a lot less computations if I negate the nands on the fly, as I build the binary tree.

One particularity of the algorithm is that it always adds '%' as the right child to the current node and keeps going deeper in the tree in this fashion until the last node is reached. Every node will always have 2 children, unless it is a leaf and won't have any. Before the last step, no '%' is added as the right child. Instead, the algorithm waits for the next, final, iteration to add the right child. Which is either a predicate node, or the root of the subtree representing the a row in the table.

An interesting sidecase is when the input has only one true expression value as a result. As my stopping condition is that I process all the true rows and then add the last child as the right child, the algorithm doesn't get to add the child to its left child as well. An additional check has been added for this.
Finally, a function converts the tree data structure back to a string. Something worth mentioning is the fact that initially I wanted to compute the nand starting from a full dnf, but this proved impossible in practice as the UI would become frozen, even if the computations were performed. This introduced an error, as on some cases, the algorithm does not get to fill all the nodes for one line of the table since the stopping condition asked for the last predicate value in the row, which in case of simplified table could be a 'star'. The value would have been ignored and tree would have been missing predicate leaves. I solved this by getting the number of non-star values in the row and compare against it a count of the number of non-star elements already processed.

Software design

I think the first thing that has to be argued is the choice of the programming language, Python. First of all, I have to say that an important part of this decision was influenced by the setup I already had on my computer and also by the OS I use which is a Linux distribution. Language specific to Windows such as C# were therefore out of the question. A good alternative, maybe just as good as the platform requirements go, was Java. However, I opted for a programming language which is more back-end related and does not require heavy IDE's such as Netbeans or JetBrains, but just a text editor and a command line. As far as the

actual programming language goes, I like Python because it has a very straightforward syntax, does not enforce unnecessary constraints on the user such as wrapping every piece of code that might raise an exception and gives more control to the programmer overall. Needless to say that this does not mean the programmer does not have to abide certain standards. But it is generally less verbose and more flexible. It is also one of the programming languages I would like to work in, besides C or C++, and does not present the memory management difficulties that these introduce. So, the choice of the programming language is more of a personal preference, but I believe it is relevant as it impacts the software design.

The application has a main function that sets up the user interface as well as the event handlers for the GUI components. There are 2 entry points in the application, as triggered by the user: analysis of the input expression and all the computations that come with it, and running of the tests. These 2 entry points are completely independent for the obvious reason that tests are not influenced by user input.

The analysis entry point does the following things in order: validates input, parses input and displays binary tree of the expression, generates truth table and simplified truth table, computes dnf and simplified dnf and computes nand.

For each one of these functionalities a number of methods have been created. Input is validated in almost every function. The idea is that functions that belong to the same functionality can be seen as standalone blocks even if for convenience (meaning readability as well as testing purposes) they have been split, they occur in predetermined places in the application. So in the case
of a function making use of some others, validating the input in the former guarantees validated input in the latter, given that the latter is not called
in any other places in the code. This is the case of computing the simplified table

functionality.

For holding the data of each symbol in the expression, a Node class was created. This class represents the building block of the binary tree that represents the input expression.

This class holds data such as the ASCII representation of the logical operator/name of the predicate, reference to one or both of its children. It also has a label that uniquely identifies it and a value attribute that represents the truth value the node has at some point in the computation.
The idea of having separate classes for a unary, binary operator and predicate and having them inherit from the Node class seemed to me completely unnecessary, as there is nothing about each one of these that has different from the others. The only other class created is the one holding values for the truth table displayed on the UI and also has the responsibility of initializing the UI components. Other than that, I tried to keep functions as modular as possible, using intuitive naming and avoiding obfuscation or abbreviation of the code altogether except maybe for a couple of words (such as ‘to’ to ‘2’, ‘dictionary’ to ‘dict’, ‘predicate’ to ‘p’, ‘disjunctive normal form’ to ‘dnf’, ‘vertical scrollbar’ to ‘vsb’). I imagine a more object-oriented approach would have been to initialize a class instance, set some attributes and call methods based on those attributes. However, my intention was to have function computations perform independently

of one another, so I would not have to create additional objects just to get values out of functions that belong to different states.

A separate mention deserves the main entry point of the expression analysis, which has not been mentioned in any of the sections above.
It provides a case distinction between a one and a multi-character input after the expression has its spaces trimmed off. In the case of a multi-character input, specialized function is called which compares the input against a regex unary or binary operator pattern and calls recursively if the condition proves to be true. If the input is just one character, and that is a letter, than all the computation is done in the main entry point of analysis, without any additional call.

Functions have been distributed over modules that represent different functionalities (actually, they describe the assignments of every week) for improved code readability and a better structure. Documentation was provided for every function and every test or fixture and it can be found inside the body of each function. There you can find a short description of what the function does, the significance of its parameters and return values.

GUI

The UI is minimal and it is written from scratch. That means that every element including its position and content is not initialized from automated code-creation tool.

At startup it holds only 3 elements: a text box where the user input is expected (the expression to be analyzed), and 2 buttons. One of them parses the expression given as input while the other one runs the unit tests, independently on the input expression. You would be able to see on the UI a message specifying whether the tests passed or not, or if another error occurred according to the exit code of pytest. For more information on testing, see the next section.

After an expression was parsed, the UI displays the computations: dnf form of the expression, the simplified dnf form, the nand form based on the dnf along with a full truth table and a simplified truth table. Each of these elements have their own identifiable labels. The parsed binary tree is displayed in a new window in a image viewer that depends on the used OS and available programs. In case of an empty or erroneous input, the application ignores it.

As far as the type of elements I chose to display the computations in that is a label. It seemed to me as being the most natural choice and definitely the kind of component whose purpose it to display information that does not come in the form of a list or a table. The truth tables are obviously displayed in some tables (technically speaking, tree views, but in this case they look just as tables).

A disclaimer should be made about the UI: the application was written on a different operating system than the one it is assessed on; it is possible that certain elements might not fit in correctly or be placed in the same way as it was seen at the design time. The application was run on both Windows 7 and Ubuntu with some small differences in how the UI looked like.

Testing

Tests are provided for every written function. Each function is tested against null or missing input and occasionally other sidecases are also taken into consideration. Validation of the input is done at the entry point in the application, as well as in any other function at any level in the function call tree.

For the purpose of proving the functionality of the application, only unit tests have been considered. Below I will target each functionality along with its tests and mention what cases have been taken into consideration for each one of them. Testing goes as early in the function call tree as possible, testing every function starting (but not including) the main entry point for parsing the input, for reasons that are tied to initializing the UI and the impossibility of finding a framework that automates end-to-end UI testing.

The only exception, besides the UI-related functions, is represented by functions that perform rendering the tree in using the graphviz framework. They represent just an interface to the used API and does not perform any computation that would be worth testing. The main issue with testing them would be that it would be generating many parsed images, which for convenience I chose to avoid.

First of all, the testing code contains fixtures which generally have the purpose of setting up and tearing down (not the case) the test code. I used them to provide values that would be used by multiple tests, along with the parametrization of the test methods. I also created 2 helpers: one for assessing whether 2 trees are the same and one for checking whether a tree has its leaves initialized for computation. I will not mention all the fixtures used, but I will briefly mention reasons they were used for: to provide list of nodes, binary tree, dictionary that stands for truth table. They can be found in the conftest file.

Technical detail: letting pytest know where it can find the source code of the app, you must change the PYTHONPATH=%PYTHONPATH%;path\to\main\folder (in Windows). This is the only thing that has to be done for the tests to run at all.
For choosing which tests to run, you may use the terminal (as long as you have pytest installed):

pytest -k ‘string to match against test functions’ or

decorate functions with:

@pytest.mark.wip

for example, and run them with:

pytest -k wip.

Pytest can be installed as easy as:

pip install pytest (or pip3 for that matter).

As far as the test functions go, they are as follows:

test_convert2expression: tests converting tree to ASCII expression. Different cases are taken into consideration: one with normal input, one that has only a root and one where the tree is null and the output should be empty.

test_convert2tree: tests converting ASCII expression to tree. A case where the input is just an alphabetical character is taken into consideration. Function otherwise returns null on null input.
test_build_line_dnf: tests building a line dnf. Takes into consideration standard input as well as 1 element, empty and null input.
test_build_full_dnf: tests building a full dnf given line dnf's. It takes into account normal, one element and null input.
test_compute_dnf: tests full dnf computing functionality. It takes into account standard input as well as several combinations of wrong input that should lead to an empty dnf.
test_compute_simplified_dnf: Tests full compute dnf functionality. Takes into consideration standard, one-element, missing and null input.
test_computing_nand: Tests computing nand. Besides standard input, it takes into consideration empty or null input.
test_create_table: Tests creating a truth table. Normal, missing or null input is taken into consideration.
test_initialize_tree: Tests initializing the leaves of a tree. Combinations of missing/null input are also checked.
test_compute_node_value: Tests computing the value of a node based on its children. Cases for all of the used operators are taken into consideration, including combinations of missing/null input.
test_compute_result: The test checks for a case where the output of the tested function is compared against what the result should be, given that the truth table is the one supplied by the fixture. In that case, the test checks for failure, since the tested function rereferences the truth table to a complete one (which is definitely not the case of the fixture). The other case considers the output for all the 2^n rows from a table, given that n is the number of predicates. In this case, the test checks for a pass. The test also considers cases of missing/null input.

test_check_stars: Tests for 2 cases where the output should be False, and a case where the row is not full of stars. It also takes into consideration missing and null input.
test_check_table_simplified: As the name suggests, tests for checking whether the table can be simplified. A case where it is possible and a case where it doesn't are provided. It also takes into account missing or null input.

test_append2dict: Tests different cases of appending rows to a dictionary truth table. A standard case, a case where the row is empty and one where the row is null are taken into consideration. The function also checks for null dictionary.

test_generate_expansion: Tests for a case where the simplified row is valid, one in which it's not and some sidecased where the input is null or missing.

test_get_true_rows: Test getting the indexes of rows that have an expression result value of 1. The test takes into consideration 2 standard inputs as well as missing or null.
test_compute_intermediate_simplified: Tests computing an intermediate simplified table. This function does not test for missing or null input, as it is

called only from one place in the application, and that is from the main simplification function.
test_compute_simplified_predicates: Tests the main simplification function. Function takes into account 2 standard cases as well as different combinations of missing/null input.
test_regex_validate: Tests the input validation function. It is done by taking into account missing/null input as well.
test_generate_hash_code: Tests include 2 normal cases as well as missing/null input.

Conclusions and future implementations

I'm fully aware of the fact that the application is not perfect and there are always things to improve. However, I think that I managed to take into consideration quite a number of aspects while writing this code. I think that a strong point of this application is the input validation and as far as the design goes, I think that the inline code documentation is also a plus.
Overall I have the impression that my algorithms didn't fully capture all the possible cases from the start and I ended up patching each time a side case appeared. I tried to this to the best of my capabilities and time resources, but I think it is possible that there are still cases that the application doesn't handle. I would say that if I had to write this application again I would try to get a bigger picture on the all the possibilities that might exist, but this is not something that I did not try; it just proved that there are so many moving parts and condition checks that have to be coordinated that many things can go wrong.

As far as improvements go, a possible one would be to have computing the normal and the standard dnf in a single function. In terms of code- design it would be to use a class for each functionality since it will abide more the object-oriented standards, as well as delegating the computing value of a node function to the Node class.

The code can have improved readability by giving up on computational complexity. I think this would have made the code readable and more easier to split. Thus, much easier to observe and debug.
Also, maybe the tests should have been checking for a bigger number of possible inputs.