Coursework 1 Report

Adi Yerembessov, 20160740

April 5, 2018

1 Conjunctive Normal Form

1.1 Data structure

I converted the given string into an expression tree by using tree_construct() function. The nodes of the tree are instances of the class Node. A tree can be accessed by having a variable pointing to the root of the tree and self.left, self.right attributes. Also a node having char '-'(negation) is supposed to have exactly 1 child, which is self.left. A node, representing a literal, is supposed to have no children. A node, representing other operators, is supposed to have 2 children.

1.2 Conversion to conjunctive normal form

After converting the string into an expression tree, I used the tree for other functions, which are used to produce a formula, equivalent to the given one, in CNF. Those functions are just implementations of the algorithms described in the lecture Propositional Logic: Normal Forms.

1.3 CNF of the given formula in Polish and infix notations

In order to print out the produced formula in Polish notation, I did pre-order traversal of the tree, and concatenated the chars in the nodes while traversing. In order to print out the given formula in infix notations, I did in-order traversal of the tree, and concatenated the chars in the nodes, however when a node with '&' operator was visited, parenthesis were added. After the traversal there could be extra parentheses, which are deleted by ensuring that there are no several parentheses in a row.

1.4 Validity

Even though there was described an efficient way of deciding validity of a formula in CNF, I chose a way, which was easier to implement, since everything was already implemented for it. In order to decide whether the given formula is valid, I used the tree of the formula in CNF, attached negation before the root. Thereafter, the negation node became the root and the tree represented negation of the given formula. And the

formula was implication free, since CNF formulas don't have implications. Then, I used negation_free() and cnf() functions to convert the formula into CNF, parsed it and fed it to minisat. If minisat says that UNSAT, the given formula is valid, otherwise it's not.

1.5 How to execute

I used python3 and installed minisat via brew install minisat. In order to execute the file you need to enter cnf directory via terminal and type:
$ python3 cnf.py "given formula"

2 Nonogram

2.1 Key idea

The idea is to find all acceptable configurations for a row/column by acceptable_options() function, remove those from all configurations produced by all_options(). Then we have all unacceptable configurations for a row/column, those configurations are connected by or operators, which is not CNF yet. Now if we negate the pile of configurations we get constraints for acceptable configurations, since those constraints guarantee that we can't get an unacceptable configuration. Most importantly, due to De Morgan's law they are connected by '&' operators, which is CNF. Note that acceptable_options() function produces mapped acceptable configurations, where 0 are mapped to 1's, 1's are mapped to 0's, so that we don't have to map final configurations after removing unmapped acceptable configurations while applying De Morgan's law.
This procedure is done for every row and column and connected by '&' operators. Now we just feed the constraints to minisat and get the solution.

2.2 Possible improvements

There are few optimizations I have in mind. The first one is using numpy. The second one is implementing all_options() by using binary representation of a number, i.e n = 0, increment n in a loop for 2^{length} times, where length is length of a row/column, in the loop convert the binary representation of n into an array, add as many 0's as needed and put it into all options. This way it'd take O(1) to remove an acceptable configuration, since index of the option needed to remove can be calculated by converting the acceptable configuration into a binary number and then converting the number into a decimal one. However, it doesn't affect capability of the tool to solve Nonograms, since I've put some Nonograms, which were fed to minisat(took reasonable time to create constraints), and weren't solved by minisat in at least few minutes. Additionally, this project already took way too much time.

2.3 How to execute

In order to execute the file you need to enter Nonogram directory via terminal and type:
$ python3 Nonogram.py "name of the cwd file"