v0.3

CS 5112 Algorithms and Data Structures for Applications Fall 2019

Homework 3

TA: Gengmo Qi *Due: November 20, 2019, 11:59 PM*

In this assignment, you will implement a polynomial-time reduction yourself.

Key instructions:

- 1. Collaboration: You are **required** to work in groups of 2 students on each assignment. Please indicate the name of your collaborator at the top of each assignment and cite any references you used (including articles, books, code, websites, and personal communications). If you're not sure whether to cite a source, err on the side of caution and cite it. Remember NOT TO PLAGIARIZE: **all solutions must be written by members of the group**.
- 2. Partner-finding: You have one week to find your preferred partner yourself and form your own group **on CMS**. If you have not formed a group on CMS by the **partner-finding deadline on 11.13th 11:59PM**, we will run a partner-matching script to assign groups for you.
- 3. Please only use **Python 3** for this assignment [1]
- 4. Please do NOT use any additional imports, only write your code where you see `TODO: YOUR CODE HERE`, and change your return value accordingly.
- 5. Reminder on Late Policy: Each student has a total of one slip day that may be used without penalty for homework. We will also drop your lowest homework score. An assignment can be at most one day late without penalty via slip days.
- 6. Please modify and submit the following files: `sat_solver.py`

---

1 Reduction

1.1 "Is it turtles all the way down?"

In lecture 11, we talked about how one can use the powerful tool of reduction to show that the Traveling Salesman Problem(TSP) is NP-complete **assuming that** we already know Hamiltonian cycle is NP-complete. However, how do we know Hamiltonian cycle is NP-complete in the first place? What's the beginning of this chain of reductions?

Definition 1.1 *(informal)* ***SAT(Boolean satisfiability problem)*** *asks whether the variables of a given Boolean Expression can be replaced by the values TRUE or FALSE in such a way that the expression evaluates to TRUE. If this is the case, the expression is called satisfiable.*

[1] our beloved Python 2 is retiring: https://pythonclock.org

For example, the expression $A\& \sim B$ is **satisfiable** because one can find the values A = TRUE, B = FALSE, which make $A\& \sim B$ = TRUE. In contrast, $A\& \sim A$ is **unsatisfiable** because this expression is FALSE for **all possible** variable assignments. [2]

[Cook, 1971] and [Levin, 1973] proved that SAT is NP-complete, and that is the first problem proven NP-completeness.

1.2 How might we proceed then?

Since the SAT problem is NP-complete, only algorithms(solvers) with exponential worst-case complexity are known, bad news. One idea to make progress is, can we reduce the users arbitrary-looking Boolean Expression to a simpler notation/form? That would simplify the solvers job.

Definition 1.2 ***CNF: Conjunctive Normal Form*** *A Boolean Expression is in conjunctive normal form (CNF) if it is a conjunction(AND: &) of one or more clauses, where a clause is an OR(|) of literals; otherwise put, it is an AND of ORs.*[3]
**Form:*

- ***CNF Expression*** *= AND (&) of clauses, e.g.*$(A|B)\&(C|D|E)$ *is a CNF Expression with 2 clauses*
- ***clause*** *= OR (|) of literals, e.g.*$(A|B|C)$ *is a clause of 3 literals; A alone can also be a clause with just 1 literal*
- ***literal*** *=* A*(positive literal),* $\sim A$ *(negative literal, meaning NOT A)*
- ***symbol*** *=* A*,* B*,* P*,* Q *etc.* [4]

CNF examples: All of the following expressions in the variable symbols A, B, C, D, E, and F are in conjunctive normal form:

- $(A|B| \sim C)\&(D|E|F)$
- $(A|B)\&C$
- $A|B$ (viewed as a conjunction with just one clause $A|B$)
- $A\&B$ (viewed as a conjunction of 2 single-literal clauses, A, B)
- A

non-examples: The following expressions are **not** in CNF:

- $\sim (B|C)$, since an OR is nested within a NOT
- $(A\&B)|C$
- $A\&(B|(D\&E))$, since an AND is nested within an OR

However, **every Boolean Expression** can be equivalently converted into CNF. For the three non-examples just mentioned, they are respectively equivalent to the following three CNF Expressions:

- $\sim B\& \sim C$
- $(A|C)\&(B|C)$
- $A\&(B|D)\&(B|E)$

[2]Why study SAT? The decision problem of SAT is of central importance to fields like Artificial Intelligence(constraint solving, planning), cryptography etc. not only in theory, but also has huge impacts in real-life softwares (config management-inside your favourite IDE, model checking etc.)

[3]This is equivalent to the POS (Product of sums) form if you are familiar with digital electronics.

[4]In our homework Python code we restrict the symbols that represents Boolean variables to be uppercase, so 'a' cannot be used but 'A' can to form an Boolean Expression

v0.3

Reduction to CNF [5] In order to reduce the SAT problem into the CNF-SAT(whose definition is very similar to SAT except that the Boolean Expression is in Conjunctive Normal Form) problem, we need to convert any arbitrary Boolean Expression into CNF, and we do so by a reduction *gadget* (the `to_cnf_gadget` in `sat_solver.py`), which conducts the following procedure[6] while making sure each step does not change the meaning of the original expression.

- Eliminate implications and equivalences:
  - `step1 = parse_iff_implies(input)`
    - $A \leftrightarrow B = (A \to B)\&(B \to A)$
    - $A \to B = \sim A|B$
- Move NOTs inwards by recursively applying De Morgan's Law [7]
  - `step 2 = deMorgansLaw(step1)`
    - $\sim(\sim A) = A$
    - $\sim(A|B) = \sim A\& \sim B$
    - $\sim(A\&B) = \sim A| \sim B$
- Distribute AND over OR, recursively
  - `step3 = distibutiveLaw(step2)`
    - $(A\&B)|C = (A|C)\&(B|C)$

Example In the python reduction gadget we are implementing, we only consider the following operators: AND(&, $\wedge$), OR(|, $\vee$), NOT($\sim$), IMPLIES(==>), IFF(<=>).

For example, the input $\sim(P \to Q)|(R \to P)$ is represented in the first row of the table below, and each row after that shows the expected return value when each step returns.

Operation	return value: Boolean Formula	return value: Python Infix
Input `s`	~(P → Q)∨(R → P)	`~ ( P | '==>' | Q ) | ( R | '==>' | P )`
parse_iff_implies( )	~(Q ∨ ~P) ∨ (P ∨ ~R)	`~ ( Q | ~P ) | ( P | ~R )`
deMorgansLaw( )	(~Q ∧ P) ∨ (P ∨ ~R)	`( ~Q & P ) | ( P | ~R )`
distibutiveLaw( )	(~Q ∨ P ∨ ~R) ∧ (P ∨ P ∨ ~R)	`( ~Q | P | ~R ) & ( P | P | ~R )`

After running `distributiveLaw()`, the expression has been reduced to CNF form:$(\sim Q|P| \sim R)\&(P|P| \sim R)$ and ready to be processed by a CNF-SAT solver.

[5] The reduction described is polynomial time. And since we have reduced the original SAT problem to the CNF-SAT problem through this reduction, we now know CNF-SAT is NP-hard. Additionally CNF-SAT is in NP so CNF-SAT is still NP-complete.

[6] [Russell and Norvig, 2009]

[7] CNF requires NOT to appear only in literals, so we "move NOT inwards" repeatedly

v0.3

1.3 Notes on the Python implementation

operators Here is a table of the operators that can be used to form Boolean Expressions. Note that the syntax for implication and equivalence is a bit funny but we are using it to overload the operators in Python.[8]

Operation	Math	Python Infix Input	Python Output	Python `Expr` Input
Negation	$\overline{P}$	`~P`	`~P`	`Expr('~', P)`
And	$P \land Q$	`P & Q`	`P & Q`	`Expr('&', P, Q)`
Or	$P \lor Q$	`P | Q`	`P | Q`	`Expr('|', P, Q)`
Implication	$P \rightarrow Q$	`P | '==>' | Q`	`P ==> Q`	`Expr('==>', P, Q)`
Equivalence	$P \leftrightarrow Q$	`P | '<=>' | Q`	`P <=> Q`	`Expr('<=>', P, Q)`

Non-deterministic return values `SAT_solver`'s return value is **non-deterministic** and **non-exhaustive** when its input expression is satisfiable, it will only return **one** satisfying assignment when it succeeds [9]. The $\sim (P \rightarrow Q)|(R \rightarrow P)$ example above also shows this: there exists multiple satisfying assignments for that expression.

Todos

- In `sat_solver.py`:
  - implement `parse_iff_implies()` ,
  - implement `deMorgansLaw()`in `sat_solver.py`
  - implement `distibutiveLaw()` in `sat_solver.py`
- Once the above are implemented correctly, you should be able to run `python3 sat_solver.py` with no errors

1.4 Epilogue: SAT solvers and implications

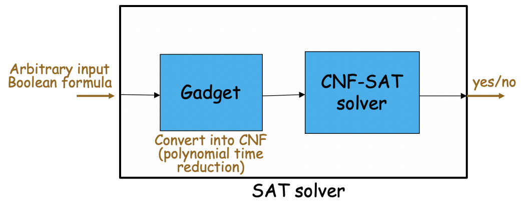
Since the SAT problem is NP-complete, only algorithms with exponential worst-case complexity are known. The good news, however, is that efficient(although exponential) algorithms for SAT do exist, among which DPLL([Davis et al., 1962]) is one of the most well-known. What DPLL does is to systematically backtrack and explore the (exponential sized) space of possible variable assignments looking for the satisfying ones.

Most SAT solvers today are essentially CNF-SAT solvers(our `dpll` as well) that forces you to enter a CNF Expression. To use an arbitrary Boolean Expression as input, you have to convert it to CNF yourself and that's why the CNF converter we are implementing is useful. [10]

[8] We do have a small problem to get around: we want to use Python's built-in operators to express the Expressions in a simple and readable way. But Python does not allow implication arrows(`==>`) as operators, so we are forced to use a more verbose notation: `|'==>'|` instead of just `==>`. Alternatively, you can always use the `Expr` constructor forms

[9] We do this just to facilitate testing, the decision problem of SAT only asks us YES(satisfiable) or NO(unsatisfiable)

[10] Some SAT solvers could be even more restrictive: They could require each CNF clause to have at most 3 literals, which means they are essentially 3-CNF-SAT solvers


Arbitrary input Boolean formula
Gadget
Convert into CNF (polynomial time reduction)
CNF-SAT solver
yes/no
SAT solver

As it turns out, reducing SAT to CNF-SAT is actually quite useful in that we can further reduce CNF-SAT to more interesting and even seemingly unrelated problems and prove their NP-Completeness[11].

2 Testing

To ensure compatibility with our grading software, please ensure that `python3 sat_solver.py` run without errors.

Just like previous homework, these tests are **non-exhaustive** meaning *passing these tests alone does not necessarily guarantee a perfect score.*

In addition to the provided tests, we encourage you to write additional tests and your own test cases. We are not providing the entire test harness for you because no one will do that for you in real-life development, you'll have to write tests yourself.

[11]see [Karp, 1972] if you are interested

References

[Cook, 1971] Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA. ACM.

[Davis et al., 1962] Davis, M., Logemann, G., and Loveland, D. (1962). A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397.

[Karp, 1972] Karp, R. M. (1972). *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA.

[Levin, 1973] Levin, L. A. (1973). Universal sequential search problems. *Problems of Information Transmission*, 9(3):265–266.

[Russell and Norvig, 2009] Russell, S. and Norvig, P. (2009). *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition.