DPLL ALGORITHM

ALI RASTEGAR MOJARAD

July 2024

Repositories

Python code and this slides:
https://github.com/4lirastegar/dpll

Web Version:
https://github.com/4lirastegar/dpll-web.git

DPLL is a SAT solver

BUT WHAT IS SAT?

It is a problem where i have some logical variables, and i have a set of clauses that are given to me or a formula is given in a conjunctive normal form(CNF).

What is CNF?

It is a way of structuring a logical formula in propositional logic.

A formula is in CNF if it is a conjunction of one or more clauses, where each clause is a disjunction of literals. A literal is either a variable or its negation.

$(A \lor \neg B) \land (B \lor C \lor \neg D) \land (\neg A \lor D)$

Literals: A, B, ¬B, C, D, ¬D, ¬A

Clauses: $(A \lor \neg B)$, $(B \lor C \lor \neg D)$, $(\neg A \lor D)$ all with disjunction(OR)

Overall formula:

The entire formula is a conjunction(AND) of all the clauses.

So this formula is in CNF.

Why SAT problems are important problems to solve?

Because it is a Canonical NP Complete Problem.

Which means, you give me a hard problem, if it is in the class NP then i can convert it into a SAT problem and i can solve it using SAT.

If i can solve it using SAT, this solution of SAT will give me the solution of the original problem.

So what's the goal?

The goal is to find assignments to variables, such that all clauses in my CNF are Satisfied.

If at least one satisfying assignment is found, the CNF formula is satisfiable(true).

Version 1 - DPLL as search

In this version we search for the satisfiable assignment.

Which takes too much time and is not sufficient.

Let see an example:

Consider a formula with this clauses is given to us:

“|” is used for OR and “!” for negation

(a | b | c)

(a | !b)

(a | !c)

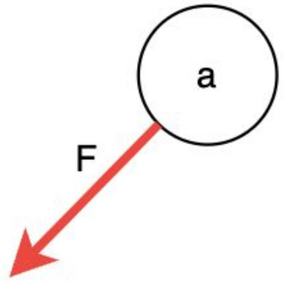
(!a | c)

(a | b | c)

(a | !b)

(a | !c)

(!a | c)

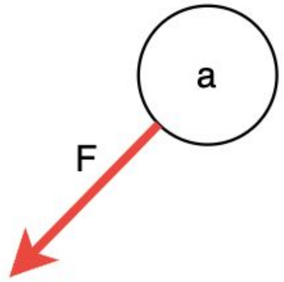


(F | b | c)

(F | !b)

(F | !c)

(T | c)

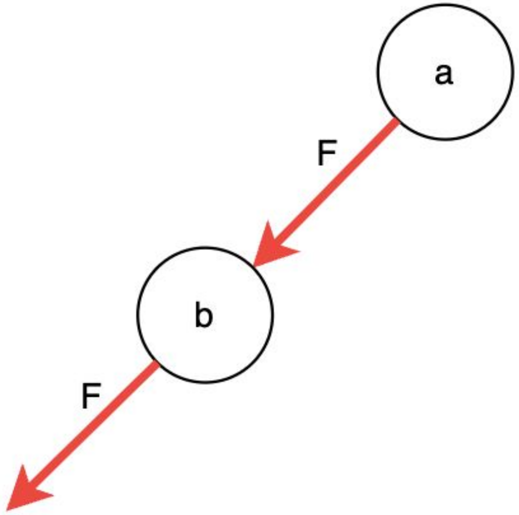


(F | F | c)

(F | T)

(F | !c)

(T | c)

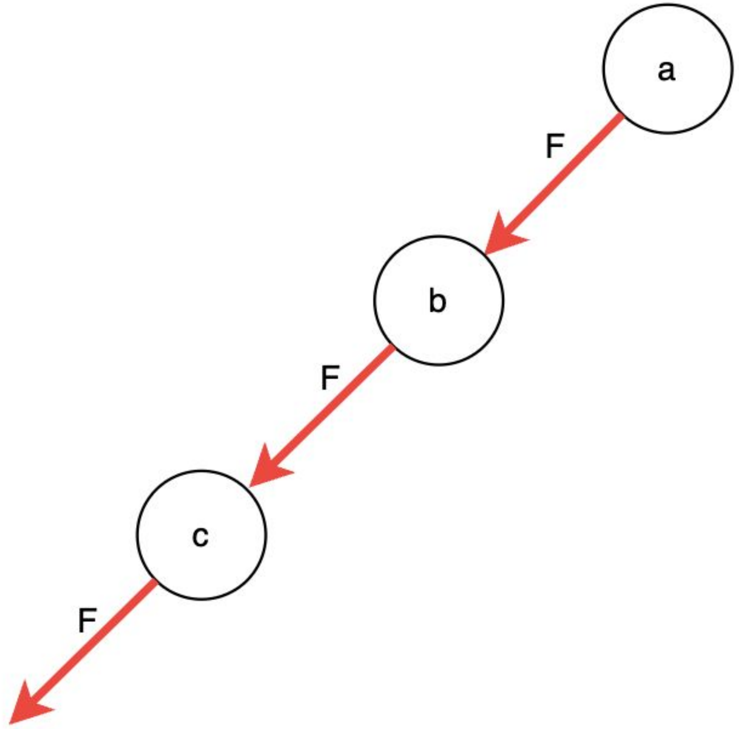


(F | F | F)

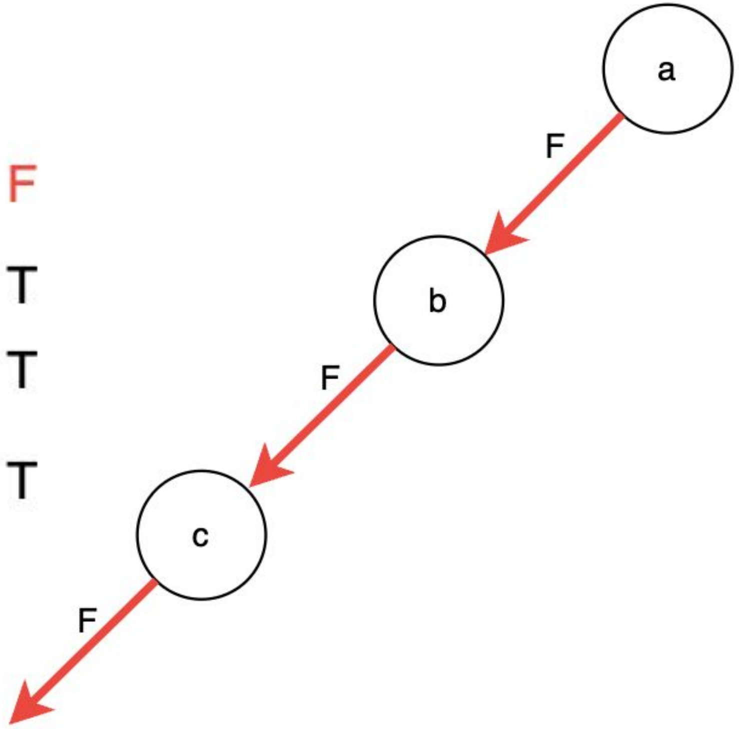
(F | T)

(F | T)

(T | F)



(F | F | F)	F
(F | T)	T
(F | T)	T
(T | F)	T

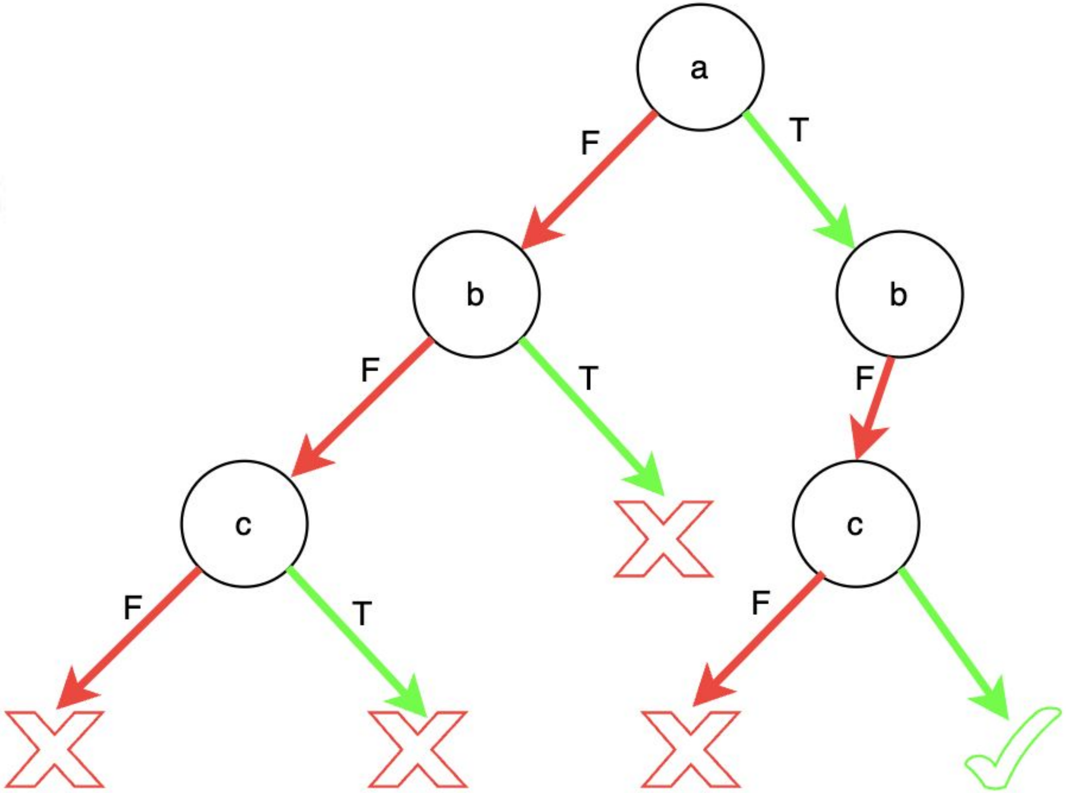


(a | b | c)

(a | !b)

(a | !c)

(!a | c)



Version 2: Improving DPLL

In this Version of DPLL we will go a step further and:

- If Literal L1 is true, then clause (L1 | L2 | …) is true
- If clause C1 is true, then C1 ^ C2 ^ C3 ^ .. has the same value as C2 ^ C3 ^..

  Therefor : Okay to delete clauses with true literals

- If Literal L1 is false, then clause (L1 | L2 | L3 | …) has the same value as (L2 | L3 | ..).

  Therefor : Okay to shorten clauses with false literals

- If literal L1 is false then clause (L1) is false

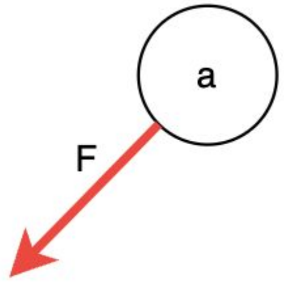
  Therefor : Empty clause is false

(F | b | c)

(F | !b)

(F | !c)

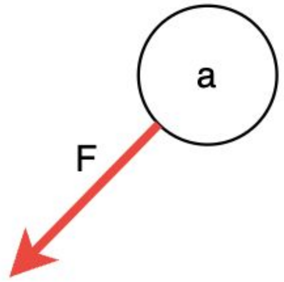
(T | c)



(b | c)

(!b)

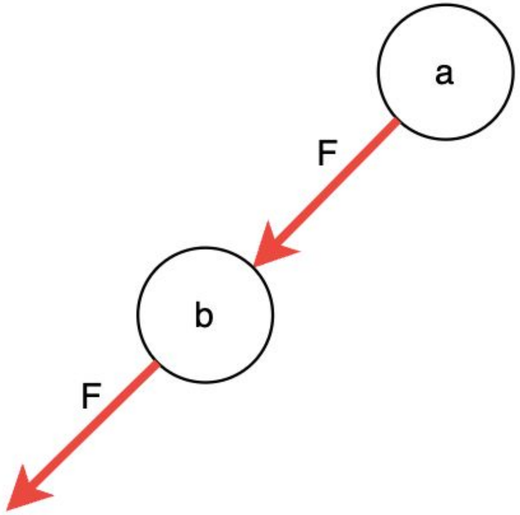
(!c)



(F | c)

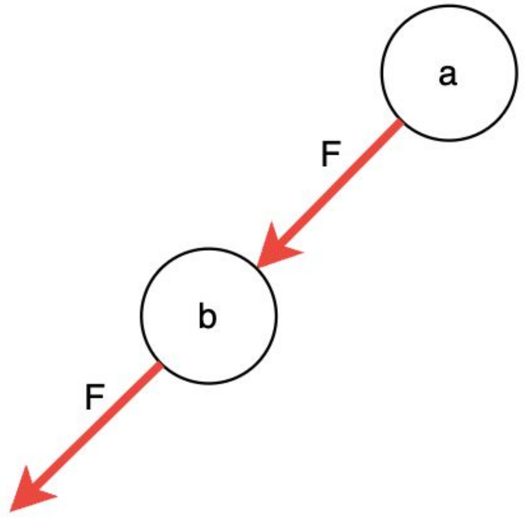
(T)

(!c)



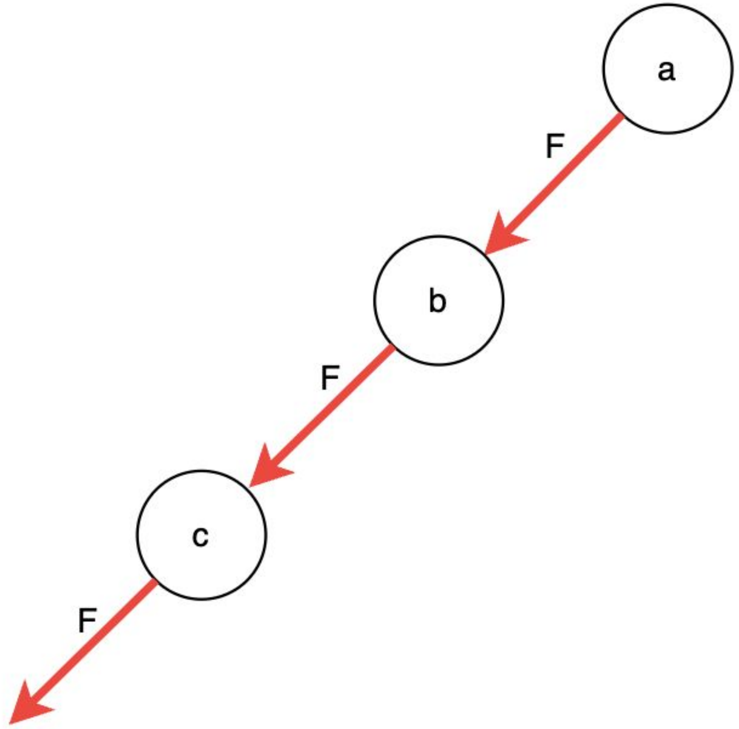
(c)

(!c)

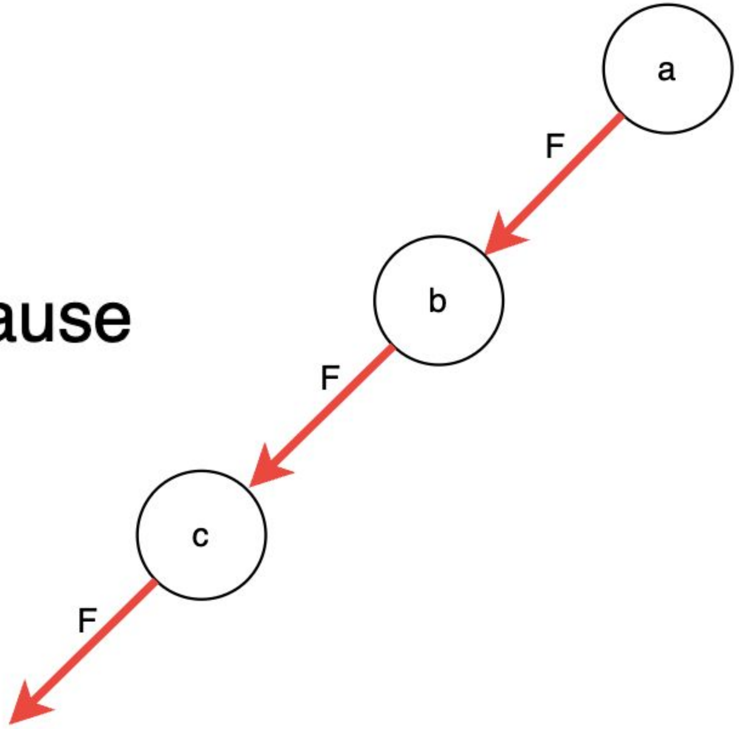


(F)

(T)



() Empty clause



Can we do better?

Yes, but how?

- Detect failures early
- Choose which variable to branch on

Look at this formula (!b c)(!c)(a !b e)(d b)(e a !c)

Looking at this formula we immediately understand if the formula wants to be satisfied c should be False, otherwise clause (!c) would be false and all the formula would become false.

So, in every clause c should be False.

This is Called **Unit Literals (Unit Propagation)**

(!b) (a !b e)(d b)

Then, set b to False, and we remain with : (d)

And then we set d to True which makes the whole formula True, and Done

What about this formula: (a !b c)(!c d !e)(!a !b e)(d b)(e a !c)

We do not have unit literals so what can i do?
If we see carefully we can see that d, in the entire formula has the same form(positive form), and we do not see it in the negated form, so if i set it to true what happens?
Do i hurt the formula? NO!
As we know we are searching for one satisfying assignment, so there may be solutions where d is False, but we don't care. WHY?
Because with setting d to True we are helping two clauses in that case and not hurting the formula, so let's do this.

This kind of literals are called **Pure Literals**

So what happens to the formula: (a !b c) (!a !b e) (e a !c)

Two clauses have become True.

Then what we do?

We set b to False, because it has the same characteristics and it is a pure literal.

And we are left with: (e a !c)

So our solution can be:

d = True, b = False, a = True

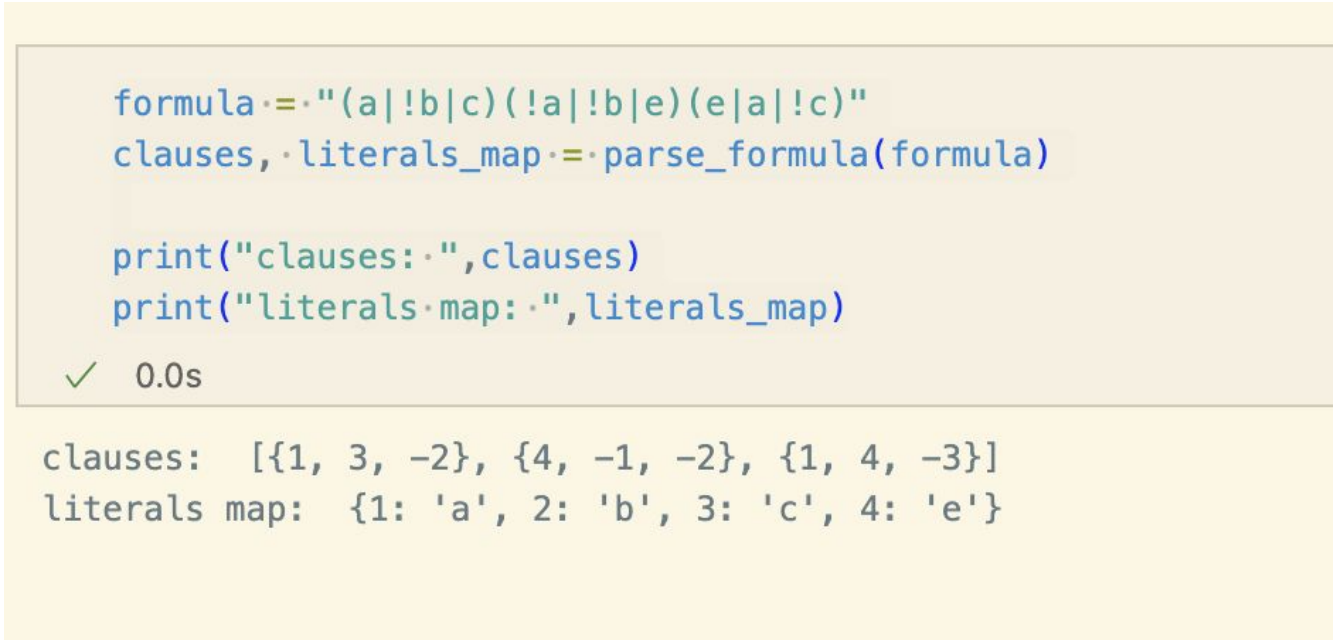
Davis-Putnam-Logemann-loveland (DPLL)

The Davis-Putnam-Logemann-Loveland (DPLL) algorithm is an essential tool for solving the Boolean satisfiability problem (SAT). It systematically determines if a propositional logic formula in Conjunctive Normal Form (CNF) is satisfiable. The DPLL algorithm enhances the basic search method with:

- **Unit Propagation**: Simplifies the formula by assigning values to single-literal clauses.
- **Pure Literal Elimination**: Removes literals that appear with only one polarity across all clauses.
- **Recursive Backtracking**: Explores possible variable assignments to find a satisfying solution.

Implementation

In the First step we get the Formula as the input like :



```
formula = "(a|!b|c)(!a|!b|e)(e|a|!c)"
clauses, literals_map = parse_formula(formula)

print("clauses: ",clauses)
print("literals map: ",literals_map)
```

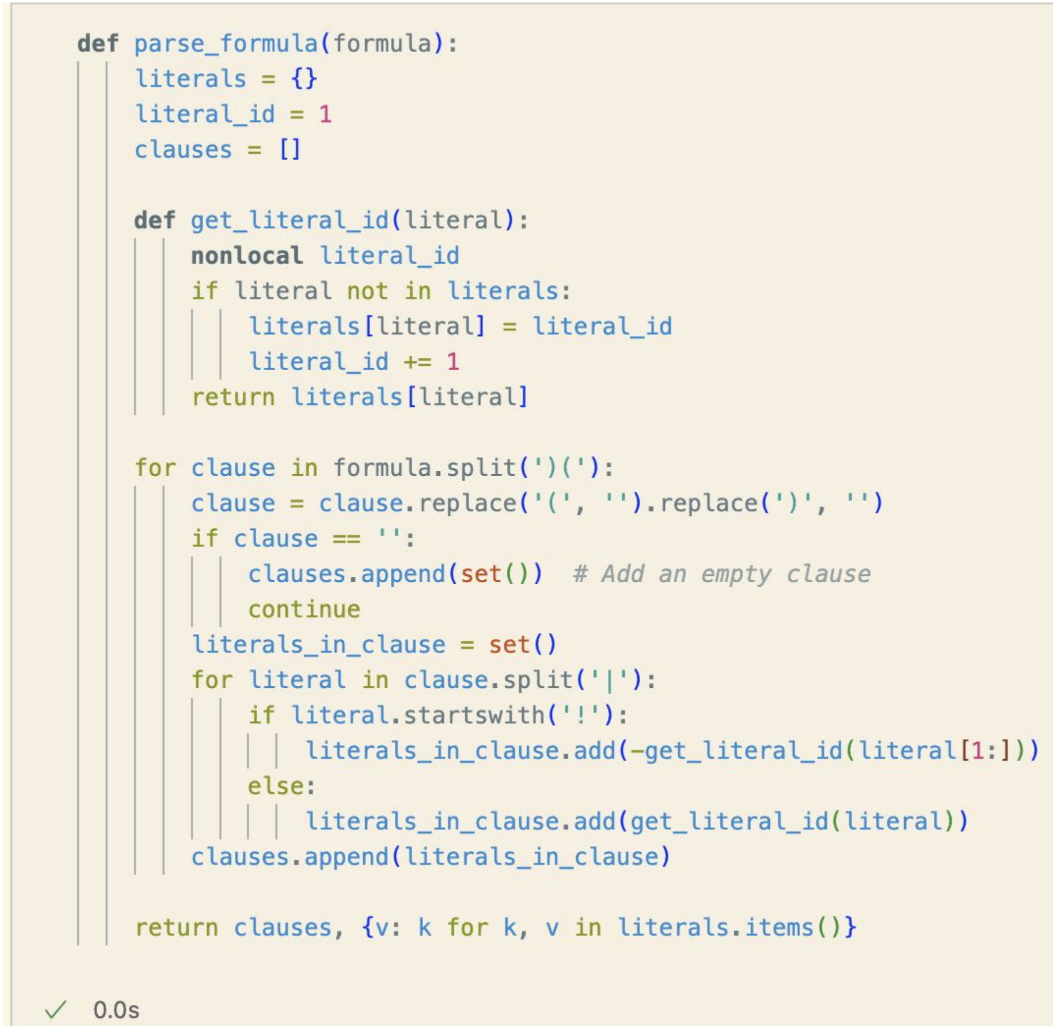
✓ 0.0s

```
clauses:  [{1, 3, -2}, {4, -1, -2}, {1, 4, -3}]
literals map:  {1: 'a', 2: 'b', 3: 'c', 4: 'e'}
```

Parse_formula

1- get_literal_id(literal):
for each literal it will return the id if it is in literal dict, if not, it will assign a new one to it.

2- split and clean the formula
And clauses



```python
def parse_formula(formula):
    literals = {}
    literal_id = 1
    clauses = []

    def get_literal_id(literal):
        nonlocal literal_id
        if literal not in literals:
            literals[literal] = literal_id
            literal_id += 1
        return literals[literal]

    for clause in formula.split(')('):
        clause = clause.replace('(', '').replace(')', '')
        if clause == '':
            clauses.append(set())  # Add an empty clause
            continue
        literals_in_clause = set()
        for literal in clause.split('|'):
            if literal.startswith('!'):
                literals_in_clause.add(-get_literal_id(literal[1:]))
            else:
                literals_in_clause.add(get_literal_id(literal))
        clauses.append(literals_in_clause)

    return clauses, {v: k for k, v in literals.items()}
```

✓ 0.0s

(a | !b | c)   (!a | !b | e)   (e | a | !c)

$$\begin{tikzcd}
a \arrow[d] & b \arrow[d] & c \arrow[d] & e \arrow[d] \\
1 & 2 & 3 & 4
\end{tikzcd}$$

```
[{1, 3, -2}, {4, -1, -2}, {1, 4, -3}]
```

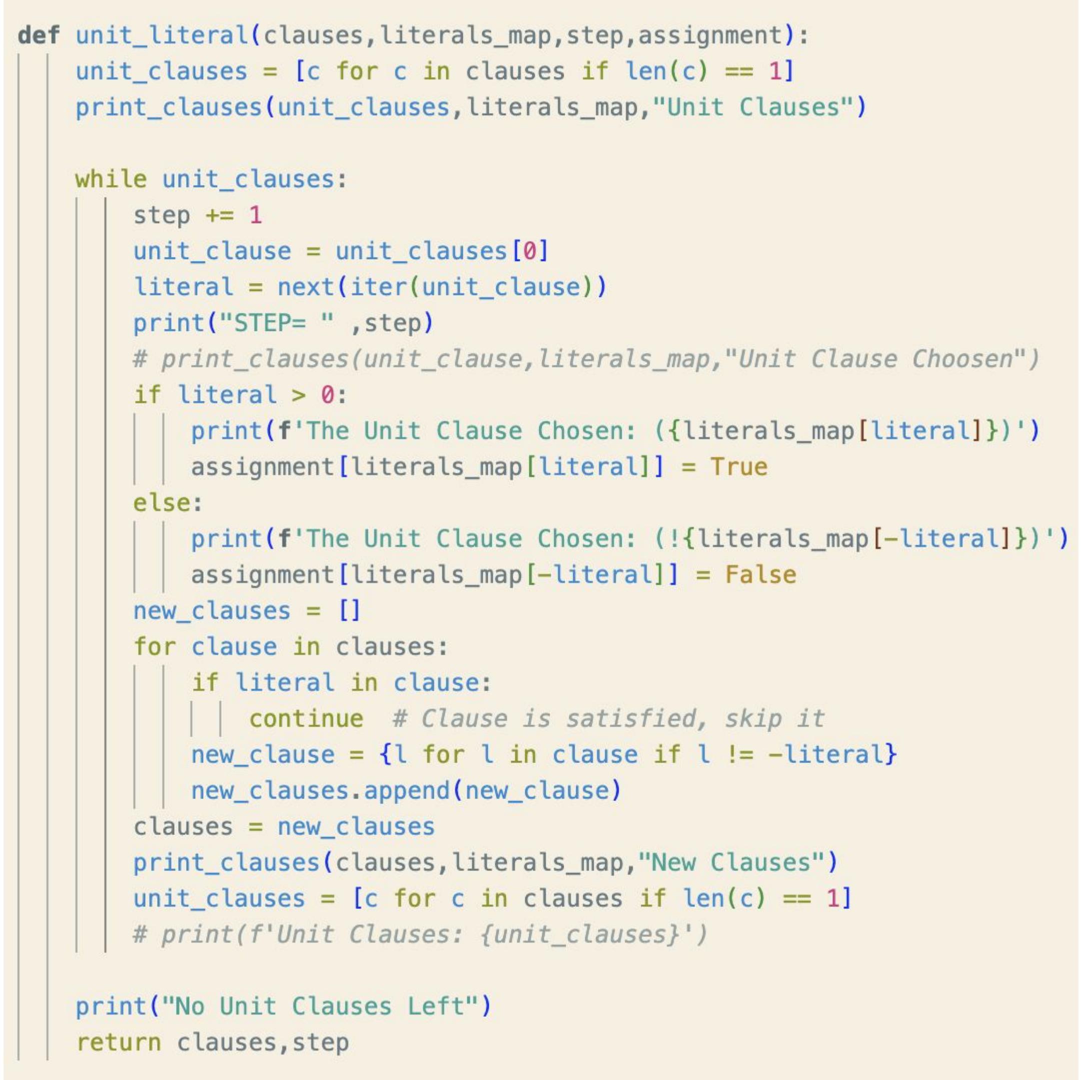
Unit Propagation

1- find unit clauses

2- choose one

3- simplify the formula

4- go to step one while there is still unit clauses



```python
def unit_literal(clauses,literals_map,step,assignment):
    unit_clauses = [c for c in clauses if len(c) == 1]
    print_clauses(unit_clauses,literals_map,"Unit Clauses")

    while unit_clauses:
        step += 1
        unit_clause = unit_clauses[0]
        literal = next(iter(unit_clause))
        print("STEP= " ,step)
        # print_clauses(unit_clause,literals_map,"Unit Clause Choosen")
        if literal > 0:
            print(f'The Unit Clause Chosen: ({literals_map[literal]})')
            assignment[literals_map[literal]] = True
        else:
            print(f'The Unit Clause Chosen: (!{literals_map[-literal]})')
            assignment[literals_map[-literal]] = False
        new_clauses = []
        for clause in clauses:
            if literal in clause:
                continue  # Clause is satisfied, skip it
            new_clause = {l for l in clause if l != -literal}
            new_clauses.append(new_clause)
        clauses = new_clauses
        print_clauses(clauses,literals_map,"New Clauses")
        unit_clauses = [c for c in clauses if len(c) == 1]
        # print(f'Unit Clauses: {unit_clauses}')

    print("No Unit Clauses Left")
    return clauses,step
```

(!b|c)(!c)(a|!b|e)(d|b)(e|a|!c)

[{2, -1},{-2},{3, 4, -1},{1, 5},{3, 4, -2}]

Unit Clauses : [{-2}]

the literal inside is < 0 so we set -literal = False
and save it in assignment

-(-2) is False
assignment={'c' : False}

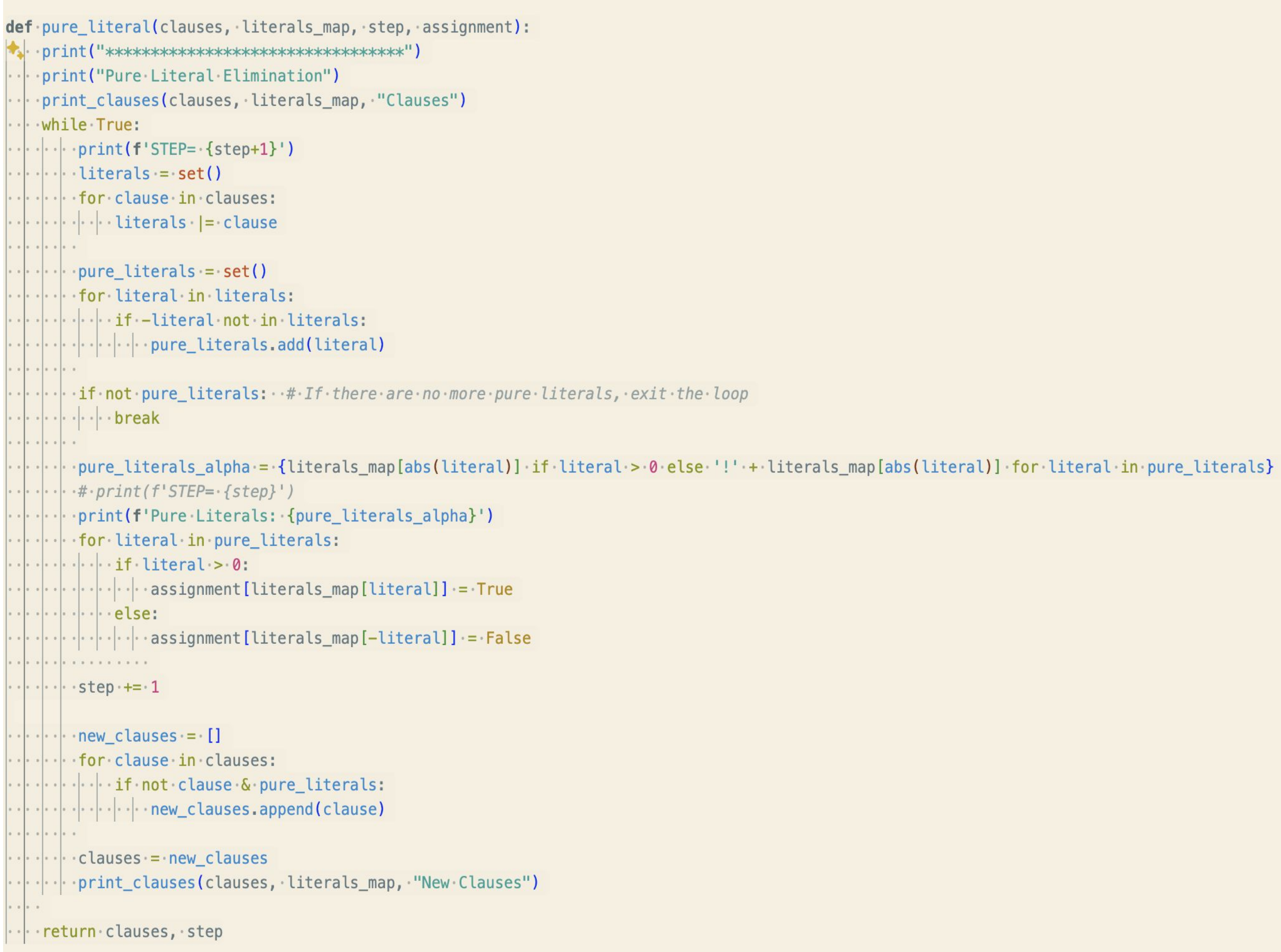
1- if literal in the clause => that clause is satisfied

2- and for every other clauses we check every literal and if the negation of our literal(False) is in it, we delete it.

```
[{2, -1},{-2},{3, 4, -1},{1, 5},{3, 4, -2}]

[{-1},        ,{3, 4, -1},{1, 5},            ]
```

Pure Literal



```python
def pure_literal(clauses, literals_map, step, assignment):
    print("*********************************")
    print("Pure Literal Elimination")
    print_clauses(clauses, literals_map, "Clauses")
    while True:
        print(f'STEP= {step+1}')
        literals = set()
        for clause in clauses:
            literals |= clause

        pure_literals = set()
        for literal in literals:
            if -literal not in literals:
                pure_literals.add(literal)

        if not pure_literals:  # If there are no more pure literals, exit the loop
            break

        pure_literals_alpha = {literals_map[abs(literal)] if literal > 0 else '!' + literals_map[abs(literal)] for literal in pure_literals}
        # print(f'STEP= {step}')
        print(f'Pure Literals: {pure_literals_alpha}')
        for literal in pure_literals:
            if literal > 0:
                assignment[literals_map[literal]] = True
            else:
                assignment[literals_map[-literal]] = False

        step += 1

        new_clauses = []
        for clause in clauses:
            if not clause & pure_literals:
                new_clauses.append(clause)

        clauses = new_clauses
        print_clauses(clauses, literals_map, "New Clauses")

    return clauses, step
```

(a|!b|c)(!c|d|!e)(!a|!b|e)(d|b)(e|a|!c)

← [{1,3,-2},{-5,4,-3},{5,-1,-2},{2,4},{1,-3,5}] →

Union of all clauses one by one to find all unique literals

{1, 2, 3, -2, -5, 4, -3, 5, -1}

Pure Literals are

{4}

if literal found is > 0 we will assign the literal to TRUE,
of literal found is < 0 we will assign -(-literal) to FALSE

Our Case => 4 or "d" is pure

we assign TRUE to d

assignment={'d' : True}

Just simply delete clauses containing pure literals

[{1,3,-2},{-5,4,-3},{5,-1,-2},{2,4},{1,-3,5}]

{1,3,-2} Intersection {4}

if Empty => keep the clause
if not Empty => delete the clause

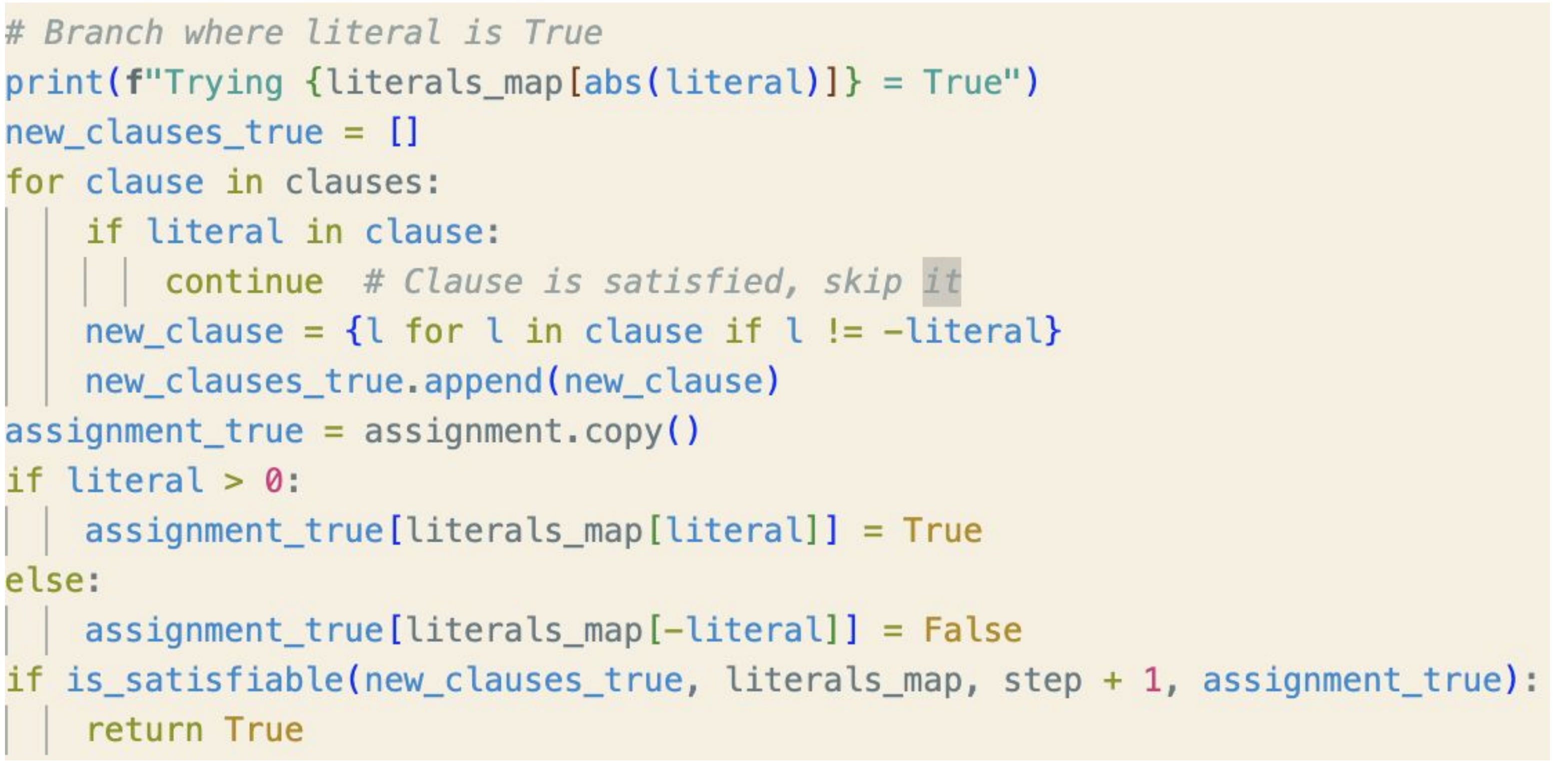
If Unit propagation and Pure litera didn't work we go for branching

- We set a variable to True and check it again with DPLL

  If it didn't work,

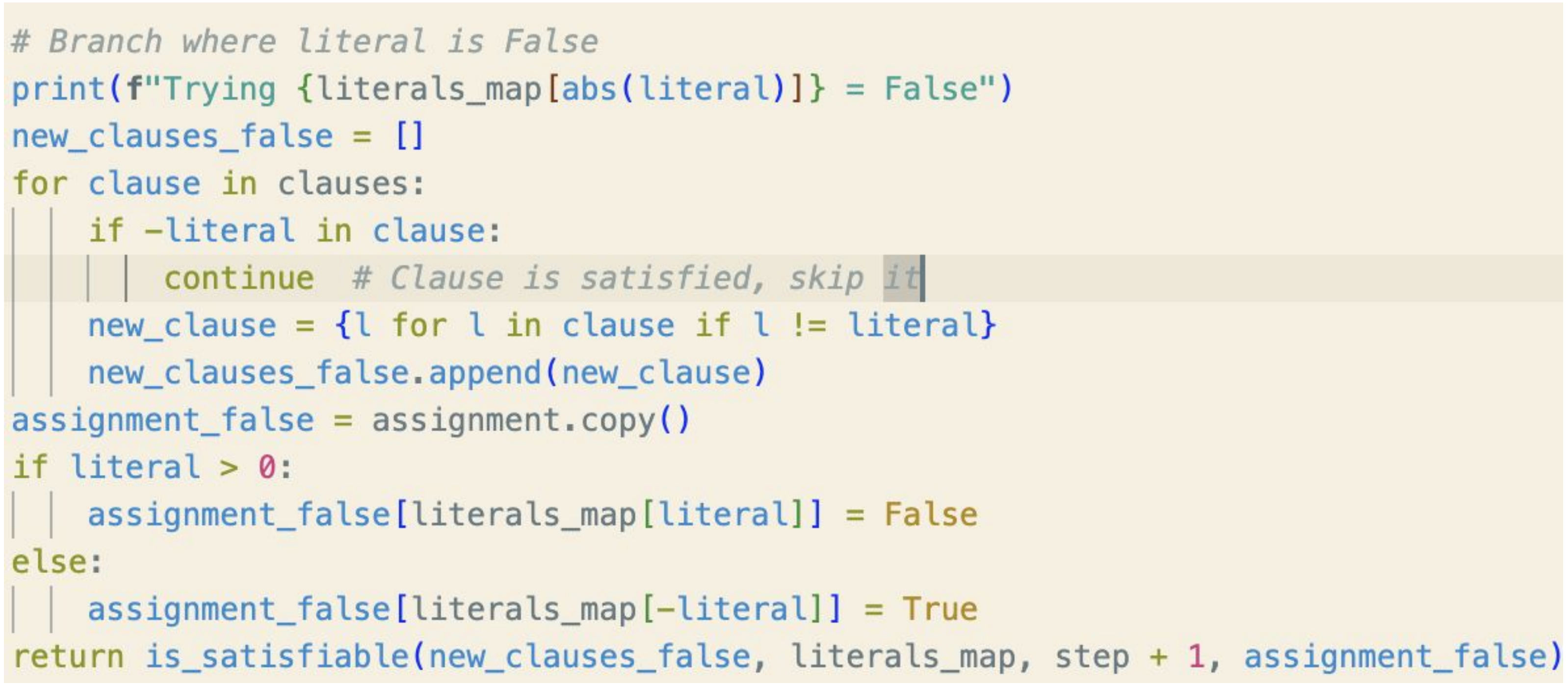
- We set the variable to False and check it again
- If both didn't work, we will go back (backtrack) and change the value assigned to the variable in the last iteration and check it again.

True Branching



```python
# Branch where literal is True
print(f"Trying {literals_map[abs(literal)]} = True")
new_clauses_true = []
for clause in clauses:
    if literal in clause:
        continue  # Clause is satisfied, skip it
    new_clause = {l for l in clause if l != -literal}
    new_clauses_true.append(new_clause)
assignment_true = assignment.copy()
if literal > 0:
    assignment_true[literals_map[literal]] = True
else:
    assignment_true[literals_map[-literal]] = False
if is_satisfiable(new_clauses_true, literals_map, step + 1, assignment_true):
    return True
```

False Branching



```python
# Branch where literal is False
print(f"Trying {literals_map[abs(literal)]} = False")
new_clauses_false = []
for clause in clauses:
    if -literal in clause:
        continue  # Clause is satisfied, skip it
    new_clause = {l for l in clause if l != literal}
    new_clauses_false.append(new_clause)
assignment_false = assignment.copy()
if literal > 0:
    assignment_false[literals_map[literal]] = False
else:
    assignment_false[literals_map[-literal]] = True
return is_satisfiable(new_clauses_false, literals_map, step + 1, assignment_false)
```

Pros :

- Efficiently reduces the problem size through unit propagation and pure literal elimination.
- Systematic approach to explore variable assignments.

Cons:

- Can still be time-consuming for very large and complex formulas.
- May require significant computational resources for worst-case scenarios.