

## Содержание

|          |   |          |
|----------|---|----------|
| <b>1</b> | <b>Постановка задачи</b>                        | <b>2</b> |
| <b>2</b> | <b>Решение</b>                                  | <b>2</b> |
| 2.1      | Описание решения . . . . .                      | 2        |
| 2.2      | Проблема и ее решение . . . . .                 | 3        |
| 2.2.1    | Проигнорировать . . . . .                       | 3        |
| 2.2.2    | Поперебирать доказательства . . . . .           | 3        |
| 2.2.3    | Сделать хоть что-то в плохом случае . . . . .   | 4        |
|          | Наивный решатель . . . . .                      | 4        |
|          | DPLL (Davis-Putnam-Logemann-Loveland) . . . . . | 4        |
| 2.3      | Структура кода . . . . .                        | 6        |
| 2.3.1    | tm.py . . . . .                                 | 6        |
| 2.3.2    | tm_iterating.py . . . . .                       | 7        |
| 2.3.3    | cnf_utils.py . . . . .                          | 7        |
| 2.3.4    | SAT-солверы . . . . .                           | 7        |
|          | Наивное решение . . . . .                       | 7        |
|          | DPLL . . . . .                                  | 7        |
| 2.3.5    | main.py . . . . .                               | 7        |
| 2.4      | Тестирование . . . . .                          | 7        |
| <b>3</b> | <b>Список литературы</b>                        | <b>8</b> |

# 1 Постановка задачи

Построить и имплементировать алгоритм, про который можно доказать следующее:

- Он распознает выполнимость 3-КНФ;
- В случае  $P = NP$  он делает это за полиномиальное время

## 2 Решение

### 2.1 Описание решения

Псевдокод:

```
Исходные параметры: Выполнимая формула  $\varphi$   
Результат: Выполняющий набор для  $\varphi$   
цикл  $n \leftarrow 1 \dots \infty$  выполнять  
|   цикл  $t \leftarrow 1 \dots n$  выполнять  
|   | Запустить машину номер  $t$  на  $n$  шагов на входе  $\varphi$ ;  
|   | если Машина  $t$  завершилась тогда  
|   | | если Выход машины  $t$  — выполняющий набор для  $\varphi$  тогда  
|   | | | Вернуть выход машины  $t$ ;  
|   | | конец  
|   | конец  
|   конец  
конец
```

Покажем, что данный код работает за полиномиальное время. Действительно, если  $P=NP$ , то существует машина тьюринга  $M$  такая, что она по выполнимой формуле  $\varphi$  выдает выполняющий набор для этой формулы за полиномиальное время. (Вообще,  $P=NP$  напрямую означает лишь существование машины Тьюринга, которая распознает принадлежность  $\varphi$  к 3-SAT, но в [1] в разделе 3.5 доказывается, что в случае  $P=NP$  можно также быстро решать соответствующую задачу поиска, чем мы и пользуемся).

Пусть эта машина работает не более, чем за  $P(|x|)$  шагов на входе  $x$ , где  $P$  — некоторый полином, а ее номер в нашей нумерации  $M$  (подробнее про реализацию нумерации и перебора машин — в разделе *Структура кода*).

Положим  $T := \max(M, P(|\varphi|))$ .

Тогда мы сделаем не более чем  $1^2 + 2^2 + \dots + T^2 = Q(T)$  шагов (под шагами подразумевается шаги эмулируемых машин тьюринга), ( $Q(x) = \frac{x(x+1)(2x+1)}{6}$  — фиксированный полином). Действительно, прежде чем переменная во внешнем цикле достигнет значения  $P(|\varphi|)$  пройдет не более  $Q(P(|\varphi|))$  шагов, а прежде чем переменная внутреннего цикла достигнет значения  $M$  пройдет не более  $Q(M)$  шагов.

Когда внешняя переменная станет равна  $P(|\varphi|)$ , а внутренняя  $M$ , то мы запустим машину  $M$  на  $P(|\varphi|)$  шагов и получим выполняющий набор (однако, возможно, мы получили его раньше и вышли).

Посмотрим, на что мы тратим время:

1. Эмуляция машин Тьюринга
2. Проверка корректности выполнимого набора
3. Итерации по циклам

Эмуляцию машин Тьюринга мы будем делать с ухудшением времени в константное число раз. Итераций по циклам никак не больше чем шагов машин Тьюринга, проверка корректности выполнимого набора выполняется за полиномиальное от размера формулы время и таких проверок будет не больше, чем итераций по циклам. Итого наше время работы можно ограничить следующей величиной:

$$(P_1(|\varphi|) + C) \cdot Q(T) = P_1(|\varphi|) + C \cdot Q(\max(M, P(|\varphi|))) \leq$$

$$P_1(|\varphi|) + C \cdot Q(M) + P_1(|\varphi|) + C \cdot Q(P(|\varphi|))$$

,  
где  $P_1$  — полином, ограничивающий время проверки корректности выполняющего набора, а  $C$  — константа ухудшения при эмуляции машины Тьюринга. В любом случае, учитывая, что  $M$  — это константа, а композиция, произведение, сумма полиномов — полином, получаем, что время работы алгоритма полиномиальное.

(Доказательством утверждений про полиномиальность проверки корректности и константного ухудшения эмуляций машин Тьюринга будет являться код, решающий поставленные задачи за заявленное время)

## 2.2 Проблема и ее решение

Внимательный читатель заметит, что мы научились находить выполняющий набор (и доказывать выполнимость) выполнимых формул за полиномиальное время (что, безусловно, радует), но ничего не сделали с невыполнимыми формулами. Моих интеллектуальных способностей хватило на три следующих выхода из ситуации:

### 2.2.1 Прогнорировать

Давайте интерпретировать условие ("Распознать выполнимость 3-КНФ за полиномиальное время") как "Понимать (доказывать), что строка является выполнимой 3-КНФ за полиномиальное время". В таком случае мы решили задачу.

### 2.2.2 Поперебирать доказательства

Давайте считать, что известно, что задача решаемая (нам же не дадут нерешаемых задач как семестровый проект, правда?). Тогда можно сделать следующее: перебирать машины тьюринга, и параллельно перебирать доказательства утверждений «Машина Тьюринга  $M_i$  решает 3-SAT за полиномиальное время в случае  $P=NP$ » (Заметим, что нельзя перебирать доказательства утверждений вида «Машина Тьюринга  $M_i$  решает

3-SAT за полиномиальное время», потому что из того, что  $P=NP$  еще не следует доказуемость этого утверждения).

Как только мы нашли машину Тьюринга, про которую существует доказательство полиномиальности ее работы, запустим ее на нашем входе и победим. Это решение полиномиально, поскольку на поиск доказательства мы потратим константу (большую, но константу) шагов.

Повторюсь, что доказуемость перебираемого утверждения я утверждаю в предположении решаемости данной мне задачи. Поэтому, в частности, это решение плохо (доказывать что-то тем, что раз мне это дали как упражнение, то это решаемое не самое математичное рассуждение). Поэтому я упомяну это решение как забавное, а сам попытаюсь перейти к чему-нибудь еще.

### 2.2.3 Сделать хоть что-то в плохом случае

Схоже с вариантом «Проигнорировать», но в отличие от него а) решает задачу в формулировке «Понять, принадлежит ли  $\varphi$  3-SAT» и б) добавляет данной работе нетривиальности.

В чем идея — давайте модифицируем основной алгоритм. Во-первых, при входе проверим, что  $\varphi$  — 3-КНФ формула. Во-вторых, по происшествии некоторого количества шагов (или просто параллельно) запустим один из существующих 3-SAT-решателей. Рассмотрим 2 основных примера.

**Наивный решатель** Перебираем все возможные наборы формул, проверяем, является ли текущий набор выполняющим. Если является — формула выполнима, вот сертификат. Если никакой набор не является выполняющим — формула невыполнима. Работает, очевидно, за  $O(N2^k)$ , где  $N$  — длина формулы,  $k$  — количество переменных

**DPLL (Davis-Putnam-Logemann-Loveland)** Вторая глава [2] утверждает, что большинство лучших SAT-решателей (по крайней мере в 2008 году) были основаны на процедуре DPLL. Это процедура, по сути, является перебором с отсечениями.

Тут применяются отсечения трех типов:

1) Вначале оценить литералы, входящие без своих отрицаний (например, если в  $\varphi$  входят только  $\neg p$ , а просто  $p$  не встречается) истинной (т.е. если было только  $p$  то положит  $p = True$ , а если было только  $\neg p$ , то  $p = False$ ).

2) В процессе перебора, если остался дизъюнкт, в котором ровно один литерал не оценен, то оценить его, если все остальные — нули (чтобы этот дизъюнкт стал истинным), и выкинуть этот дизъюнкт из рассмотрения.

3) Если мы нашли выполняющий набор, надо остановиться.

Легко видеть, что все три отсечения являются корректными.

Псевдокод:

**function** UNITPROPAGATE

**Исходные параметры:**  $\varphi$  - булева формула в КНФ,  $\rho$  – некоторая оценка переменных (возможно, не всех)  $\varphi$

**Результат:** Либо эквивалентная  $\varphi$  формула, не содержащая дизъюнктов из одной значимой переменной, либо индикатор, что  $\varphi$  тождественно ложна при такой оценке переменных

*до тех пор, пока  $B$   $\varphi$  есть дизъюнкт, в котором ровно один неоцененный литерал  $\alpha$  И не найдено противоречие* **выполнять**

**если** *Если в этом дизъюнкте все, литералы кроме  $\alpha$  оценены нулем* **тогда**

**если**  $\alpha = p$  **тогда**

|  $\rho[p] \leftarrow \text{True}$

**иначе**

|  $\rho[p] \leftarrow \text{False}$

**конец**

**конец**

Выкинуть этот дизъюнкт.

**конец**

**если** *Найдено противоречие* **тогда**

| Вернуть ' $\varphi$  – ложна'

**конец**

Вернуть  $\rho$

**function** DPLL

**Исходные параметры:**  $\varphi$  - булева формула в КНФ,  $\rho$  – некоторая оценка переменных (возможно, не всех)  $\varphi$

**Результат:** Либо выполняющий набор для  $\varphi$ , либо индикатор, что  $\varphi$  тождественно ложна при такой оценке переменных

$\rho \leftarrow \text{UnitPropagate}(\varphi, \rho);$

**если**  $\rho = \text{'}\varphi \text{ – ложна'}$  **тогда**

| Вернуть ' $\varphi$  – ложна'

**конец**

**если** *Все переменные оценены* **тогда**

**если**  $\varphi$  истинна на  $\rho$  **тогда**

| Вернуть  $\rho$

**иначе**

| Вернуть ' $\varphi$  – ложна'

**конец**

**конец**

Выбрать переменную  $x$ ;

$\rho[x] \leftarrow \text{True};$

$\text{result} \leftarrow \text{DPLL}(\varphi, \rho);$

**если**  $\text{result} \neq \text{'}\varphi \text{ – ложна'}$  **тогда**

| Вернуть  $\text{result};$

**конец**

$\rho[x] \leftarrow \text{False};$

Вернуть  $\text{DPLL}(\varphi, \rho)$

**function** Сheck3SAT

**Исходные параметры:**  $\varphi$  - булева 3-КНФ формула

**Результат:** Либо выполняющий набор для  $\varphi$ , либо индикатор, что  $\varphi$  тождественно ложна.

$\text{pre\_rho} \leftarrow \{\};$

**до тех пор, пока** в  $\varphi$  есть литерал  $\alpha$ , отрицание которого не входит в  $\varphi$   
**выполнять**

**если**  $\alpha = p$  **тогда**

$\text{pre\_rho}[p] \leftarrow \text{True}$

**иначе**

$\text{pre\_rho}[p] \leftarrow \text{False}$

**конец**

    Выкинуть из  $\varphi$  все дизъюнкты, в которые входит  $\alpha$ .

**конец**

$\text{DPLL\_Result} \leftarrow \text{DPLL}(\varphi, \{\});$

**если**  $\text{DPLL\_Result} = \varphi$  *тождественно ложна* **тогда**

    Вернуть  $\text{DPLL\_Result}$ ;

**иначе**

    Вернуть  $\text{DPLL\_Result} \cup \text{pre\_rho}$

**конец**

## 2.3 Структура кода

Пришло время вспомнить, что алгоритмы надо не только описывать словесно, но и реализовывать.

Все реализовано на языке python3, код лежит по адресу <https://gitlab.com/thefacetakt/mipt-comp-complexity-project/tree/master/code>

Для начала расскажем про код основного алгоритма:

### 2.3.1 tm.py

В этом файле содержится описание реализации машины Тьюринга и вспомогательных вещей (таких как двусторонняя бесконечная лента). Машина тьюринга, как водится, задается функцией перехода ( $\delta$ ) (ну и количеством состояний). Алфавит будет везде использоваться  $\Sigma = [0', 1', \&', '|', \sim]$  и  $\Gamma = \Sigma \cup \{\#\}$ . Стартовое состояние всегда 0, конечное всегда  $n - 1$ . Из курса логики известна эквивалентность такой машины Тьюринга обыкновенной).

Заметим, что функция *make\_step* не содержит циклов и выполняется за константное число операций, как и было завлечено выше. (Строго говоря, внутри *make\_step* присутствует вызов функции `__getitem__` класса *TwoSidedTape*, в которой может происходить расширение массива на несколько ячеек, что не назовешь константой. Однако, поскольку действие машины Тьюринга локально [то есть если мы трогаем ячейку  $i$  то на предыдущем шаге мы трогали либо ячейку  $i - 1$ , либо  $i + 1$ ] подобных линейных спецэффектов не возникает).

### 2.3.2 tm\_iterating.py

В этом файле реализована функция последовательного перебора машин Тьюринга. Как мы из перебираем: сначала перебираем число состояний (`generate_all_tms`), а внутри перебираем все машины тьюринга с данным числом состояний на нашем алфавите (то есть по сути, перебираем функцию перехода).

Из неочевидных моментов здесь может быть конструкция `yield` — она позволяет генерировать последовательность машин Тьюринга «лениво», подавая их одна за одной по требованию (приостанавливая перебор). Это очень удобно в нашем случае, ведь перебирать машин нам надо — бесконечно много. Более подробно про конструкцию `yield` и концепцию генераторов можно прочитать в официальной документации [3]

Теперь про вспомогательные функции:

### 2.3.3 cnf\_utils.py

В этом файле реализованы 2 функции: `check_3cnf` и `run_3cnf`. Первая проверяет, что строка является корректной 3-КНФ формулой (считая, что все переменные — это числа в двоичной системе от 0 до  $n-1$ , `&` отвечает за конъюнкцию, `|` — за дизъюнкцию, `~` — за отрицание. Приоритеты — стандартные), и в случае успеха разбивает формулу на дизъюнкты, а дизъюнкты на литералы.

Функция `run_3cnf` производит вычисление значения формулы (в разобранном виде) на данной оценке.

Из кода легко видеть, что обе функции работают за полиномиальное время.

### 2.3.4 SAT-солверы

**Наивное решение** Наивное решение лежит в файле `solutions/stupid.py` и делает ровно то, что и было описано выше

**DPLL** DPLL алгоритм, как несложно догадаться, лежит в файле `solutions/dpll.py` и является переводом на питон псевдокода, описанного выше.

### 2.3.5 main.py

В этой файле и реализован основной алгоритм, прямо как описано в самом начале работы

## 2.4 Тестирование

Поскольку алгоритм работает с очень очень большой константой, основным инструментом будет модульное тестирование. (Мы будем проверять на корректность программу по частям). Все тесты собраны в папке `tests`. Тесты запускаются из папки `code` командами `python3 -m tests.<название теста>` или в папке `coverage` есть скрипт `test.sh` (для его работы нужна поставленная библиотека `coverage`) Можно посмотреть покрытие тестами (`coverage/htmlcov/index.html`)

Наименование устроено очень просто – файл `tests/<name.py>` тестирует содержимое файла `name.py` (За исключением файла `tests/solutions.py`, он тестирует файлы в папке `solutions`)

### 3 Список литературы

1. Д. В. Мусатов, *Сложность вычислений. Конспект лекций* МФТИ, 2016
2. F. van Harmelen, V. Lifshitz, B.Potter *Handbook of Knowledge Representation*, 2008
3. <https://wiki.python.org/moin/Generators>
4. [https://en.wikipedia.org/wiki/P\\_versus\\_NP\\_problem](https://en.wikipedia.org/wiki/P_versus_NP_problem)