3-SAT Problem - effective algorithm

Pawel Kaminski
Poland, Europe
30 July 2026

1. Purpose of this article

In this text solution for 3-SAT problem will be shown. This is effective algorithm for solving this problem.

2. Introduction

3-SAT:
Lets have n arguments: x_1, x_2, ..., x_n. Every argument can take 0 or 1. In next step lets create boolean expression. It consist from subexpression like: x_k1 or x_k2 or x_k3 where k_1, k_2, k_3 represent values from 1, 2, ..., n. And moreover k_1 != k_2 != k_2. Another step is adding to arguments from expressions value not. It is optional. Example: not x_k1 or xk_2 or not x_k3. Now having polynomial list of such different expressions we create another one expression joining all expression with and. Example: (x_1 or x_2 or not x_3) and ... and (not x_10 or x_n-1 or not x_n)
Having such expression we try to find solution. There is 2^n possible assigments to this problem. So one algorithm to find solution to this problem is to check all 2^n possible assignments. This algorithm has exponential complexity. In presented algorihtm new easier way of solving this problem is shown.

3. Solution:

A.
Let create list of all three arguments from all arguments. There is n*(n-1)*(n-2)/6 such threes. Every three can have 2^3 = 8 assignments. Example:

x_1	x_2	x_3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

This is a list of all generated values:

```
   x_1 x_2 x_3 x_4  ... x_n
  [ 0   0   0  -1   ... -1]
  [ 0   0   1  -1   ... -1]
  ...
  [-1   0   0   0   ... -1]
  ...
  [-1 ...         -1 1 1 1]
```

So finally we have list of n*(n-1)*(n-2)/6*8 values. We notice that when having value [0 0 0 -1 ... -1] and one expression (x_1 or x_2 or not x_4) we can extend value to [0 0 0 0 -1 .. -1]. This is assumption because x_1 or x_2 is zero but positive values are searched so (not x_4) must be 0. Another step in algorithm is to delete values that make global expression 0. If we have expression (x_2 or x_10 or x_n) we can remove assignment (x_2 = 0, x_10 = 0, x_n = 0). There is n-2 such steps for all values, when we finally get possible

candidate for solutions. n-2 is needed as there is n-3 position with -1 and one another is needed for checking ptential candidate. Complexity of this step is n*(n-1)*(n-2)/6*8*(n-2)*len(3-SAT) ~ n^4*len(3-SAT). In this way we delete some values from list that do not solve the expression.

B.

But this is not everything. Given value [0 0 0 -1 ... -1] may have common one value with another expression. Example of expression is (x_2 or x_10 or x_n).

For this subexpression good values are:

x_2	x_10	x_n
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

In value we have x_2 = 0. So we now that for x_10 = 0 and x_n = 0 value have no solution. So in next step of algorithm we can pass (x_10 or x_n) as expression. We collect all such subexpressions for all possible values from set of all possible threes generated from list of arguments. So for every such value we collect specific expression. Every small expression have two or three arguments. For every value with specific list of modified expressions we run the same algorithm like in point A. But now we run this algorithm for n-3 arguments. 3 arguments we omit as we determined value for it. Every step in point B have (n-3)*(n-4)*(n-5)/6*8*(n-5)*len(3-SAT) ~ n^4*len(3-SAT) steps. For every possible three we check if it contains at least one value. If there is at least one three without values this value taken from point A can be removed.

C.

Now having all remaining values we check if every possible three from list of arguments have at least one value. If at least one three is without values it should be certain that for given 3-SAT expression no value solve this problem. If for all threes there is at least one argument we can clean final threes with values. Example is that if for three (x_1, x_2, x_3) is only value (0, 0, 0) we know that other threes containig x_1 or x_2 or x_3 must have only 0. If is 1 we remove such value from corresponding three.

D.

Now we have all threes that exists in final solution. So what we need to do is to find at least one that solves the problem. Finding all would be hard, because there can be for example 2^(n-4) solution. To find first solution we take first value from three (x_1, x_2, x_3). Now for every three with (x_1, x_2) we take passing solution. And in this step for such partial value we check if it is possible by taking all values from threes with x_1, x_2, x_3, .., x_k. At least one value from such three must fit partial value. If such partial value is not good then for x_1, x_2, x_3, ..., x_(k-1) we take another passing value from three x_1, x_2, x_k. Another step is to run whole algorithm for partialy calculated value and boolean expression that may contain subexpression with one, two or three arguments. If this step is True we know that subvalue is correct. In another case we need to take another value. So finding final solution could take maximally 2*(n-3)*[n*(n-1)*(n-2)/6*8]*[(n-3)*(n-4)*(n-5)/6*8*(n-5)*len(3-SAT)]~ O(n^8*len(3-SAT)) steps.

E.

Why two steps is needed for every algorithm. Because expression may have

one common value with candidate value. We need two steps because we know that for very three from all possible threes must contain at leats one value. If at least one threehave no values it is certain that 3-SAT have no solution and always return false. We proofe that there must be such sequence of subexpressions that excludes all eight values for given three. This is list of all possible assignments:

```
 x_1, x_2, x_3, ... , x_n
  0    0    0          0
  0    0    0          1
  ...                 ...
  1    1    1    ...   1
```

Creating 7 different subexpressions for three (x_1, x_2, x_3) we exclude seven values for this three. Then 2^(n-3) values will remain in a list. Then if the same is done for (x_4, x_5, x_6) there will be 2^(n-6) values. Doing the same for next threes there will be only three (x_(n-2), x_(n-1), x_n). This three must contain 2^3 different values that were not exlcuded.

It will be:

```
  x_(n-2), x_(n-1), x_n
     0        0      0
     0        0      1
     0        1      0
     0        1      1
     1        0      0
     1        0      1
     1        1      0
     1        1      1
```

If previous step for this three will be applied, 7 values from this three will be removed. Doing this there will be only one remaining value. Let it be

```
   x_(n-2), x_(n-1), x_n
     1        1      1
```

So one full candidate is e.g.:

```
 x_1 x_2 x_3 ... x_(n-3) x_(n-2) x_(n-1) x_n
  0   0   0   ...   0        1       1     1
```

Expression have no solution so we know that any value is possible. So we have to remove this value. If we take small subexpression (not x_(n-2) or not x_(n-1) or not x_n) then all values are removed for such three. So we know that there is one such path in algoirhtm for every value that removes them for given three.

If for (x_(n-2), x_(n-1), x_n) we take subexpression with (x_1 or not x_(n-1) or not x_n) then algorithm makes empty another three. Having subexpression with x_1 it delete remaning value (0, 0, 0) for three (x_1, x_2, x_3)

If for (x_(n-2), x_(n-1), x_n) we take subexpression with one common, value e.g. (x_1 or x_4 or not x_n) we could think that three steps of algorithm are needed. But when we consider (x_1, x_2, x_4) for two (x_1, x_2) only (0, 0) is possible because three (x_1, x_2, x_3) only have value (0, 0, 0). So only two values (0, 0, 0) and (0, 0, 1) will remain for three (x_1, x_2, x_4). Applying algorithm for (0, 0, 1) and having subexpression x_4=1 causes that remaining value (0, 0, 0) for (x_4, x_5, x_6) is removed. So (x_4, x_5, x_6) is empty set. Applying algorithm for (x_1=0, x_2=0, x4_=0) subexpress with x_n=0 exits. But this subexpression removed left (1, 1, 1) value for three (x_(n-2), x_(n-1), x_n). So empty is three (x_(n-2), x_(n-1), x_n). So algorithm makes this three empty.

If we have only one value remaining and trying delete this if any three from x_1, ..., x_(n-3) will be taken the same will be observed for other threes. Doing this for different value for given three we can leave selected three as no empty but there will be at least one other three with no values. No values means there is path of subexpressions that delete all values for given three. This is what algorithm do.

Why we dont need 3 or more nested steps of algorithm. Let consider args x_1, x_2, ..., x_7 and take subexpression (x_1 or x_2 or x_3), (x_3 or x_4 or

x_5) and (x_5 or x_6 or x_7), For three (x_1 or x_2 or x_3) and value (1, 1, 0) there will be subexpression with two (x_4 or x_5). This two have one common value with (x_5 or x_6 or x_7), Np (x_4=1, x_5=0), Then next subexpression for contains two (x_6 or x_7). But this step can be shorten taking into consideration three (x_3, x_4, x_5). Running algorithm for x_1, x_2, ..., x_7 with three subexpression mentioned above causes that algorithm does not remove anything in second step. So such dependency can be omitted in consideration.

This is the reason why we need two steps in this algorithm. Obviously we can do one step without first one. Then in second step for algorithm should be considered all values from all possible threes. Then subexpressions that suite with two arguments should be taken into consideration. Then in second step of algorithm will be subexpressions also with one argument. But having first step of algorithm we reduce some values from list earlier.

To summarize there is:

1. First step of algorithm:
   n*(n-1)*(n-2)/6*8*(n-2)*len(3-SAT) ~ O(n^4*len(3-SAT))
2. Second step of algorithm:
   [n*(n-1)*(n-2)/6*8] * [(n-3)*(n-4)*(n-5)/6*8*(n-5)*len(3-SAT)] ~ O(n^7*len(3-SAT))
3. For cleaning values there is:
   [n*(n-1)*(n-2)/6*8*n*(n-1)*(n-2)/6*8*len(3-SAT)] ~ O(n^6*len(3-SAT))
4. For finding at least one solution there is:
   2*(n-3)*[n*(n-1)*(n-2)/6*8]*[(n-3)*(n-4)*(n-5)/6*8*(n-5)*len(3-SAT)]~ O(n^8*len(3-SAT))

So maximum complexity to determine that at least one value exist or there is no solution is n^7.

So to summarize we need n^8*len(3-SAT) steps to solve the 3-SAT problem. For n=30 there is 656100000000 steps what is worse results than 2^30. For n=50 there is 2^50=1125899906842624 but 50^8= 39062500000000 what is 28-times faster result. Obviously estimation n^7*len(3-SAT) is general. len(3-SAT) is omitted as for the solution with complexity 2^n for every assigment len(3-SAT) steps is needed to check if 3-SAT have at least one solution.

Exact time complexity estimation depends on what is taken into consideration when running the algorithm. But this is not purpose of this article. In this article we presented that there is effective way of solving 3-SAT problem. Even if complexity of presented algorithm is n^10 this is much better solution than 2^n. So with presented algorithm it is possible in polynomial time to determine if any value solve 3-SAT problem. If at least one assignment solves the boolean expression it is also possible in polynomial time to determine one solution.

4. Summary

Below attached algorithm solves the sample random expression from given list of arguments. It is written in python as code in python could serve as good programming language to present general steps needed for this algorithm. Obviously it can be supposed that other programming langauges solve this problem faster. In algorithm second step is ommited as values from those calculated in first step are cleaned in other way. Algorithm can be modiified and used for commercial purpose but this modified algorithm can not ba patented.

5. Proof

The whole assumption in this algorithm is that when 3-SAT boolean expression is not satisfiable, then algorithm after second step always return three without values. So for at least one three and its all values pexits path of subexpressions that delete this values.

Proof of theorem is based on example. Let have 15 possible arguments in 3-SAT boolean expression. If 3-SAT is always false, it is known that no value is possible for given 3-SAT expression. Let take three (x_1, x_2, x_3) and value

(0, 0, 0) we can threes (x_4, x_5, x_6), ..., (x_13, x_14, x_14) show as tree with 8 children for every node except last. So (x_1, x_2, x_3) is root. For next level with three (x_4, x_5, x_6) there is 8 nodes. Next level has 64 nodes and last have 4096 nodes. If for second level 4 values is deleleted then remaining is:

(x_4, x_5, x_6), (x_7, x_8, x_9), (x_10, x_11, x_12), (x_13, x_14, x_15),
4 8 8 8

If we delete 4 values for (x_7, x_8, x_9), 4 for (x_10, x_11, x_12) and 4 for (x_13, x_14, x_15), then:

(x_4, x_5, x_6), (x_7, x_8, x_9), (x_10, x_11, x_12), (x_13, x_14, x_15),
4 4 4 4

when 3 valiues for (x_7, x_8, x_9), 3 values for (x_10, x_11, x_12) and 3 values for (x_13, x_14, x_15) will be deleted, then:

(x_4, x_5, x_6), (x_7 x_8, x_9), (x_10, x_11, x_12), (x_13, x_14, x_15),
4 1 1 1

There will be only 4 potential values left that could satisfy 3-SAT boolean expression. So for some three must exist path, that delete all 8 values. To clarify in situation above if we delete 4 values from three (x_10, x_11, x_12) for some value from three (x_7 x_8, x_9) then for other remaining values for three (x_7 x_8, x_9) this four values can be deleted. In proof we based on assumption, that sequence of 3-SAT subexpression like (x_1 or x_2 or x_3), (x_3 or x_4 or x_5) and (x_5 or x_6 or x_7) is not worth to consider. Because path (x_3 or x_4 or x_5) do (x_5 or x_6 or x_7) can be considered instead.

6. Observations

After running step A of algorithm, it can be noticed, that among remaining values exists value that does not contain -1. Then such value can be taken immediately and returned as algorithm result.

Bibliography:

1. "Introduction to Algorithms", T. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, 2022.
2. Wikipedia
3. www.jetbrains.com – PyCharm software serving as python environment for creating and testing algorithm

Problem 3-SAT - efektywny algorytm

Paweł Kamiński
Polska, Europa
30 lipca 2026

1. Cel artukułu.

W tym artykule ukazane jest rozwiązanie problemu 3-SAT. Jest to efektowny sposób rozwiązania tego problemu.

2. Wprowadzenie

3-SAT:
Jako wejście do algorytmu brana jest lista argumentów: x_1, x_2, ..., x_n. Każdemu argumentowi można przypisać wartość 0 lub 1. W następnym kroku z tych argumentów jest tworzone wyrażenie boolowskie. Składa się ono z podwyrażeń typu: (x_k1 or x_k2 or x_k3) gdzie k_1, k_2, k_3 przybierają wartości ze zbioru 1, 2, ..., n. Ponadto k_1 != k_2 != k_2. Kolejnym krokiem jest dopisanie do wybranego argumentu wartości not. Operator not dla x_1 oznacza, że dla x_1=0 wartością zwracaną jest 1, a dla x_1=1 wartością zwracaną jest 0. Przykład wyrażenia: not x_k1 or xk_2 or not x_k3. W następnym kroku trójki zostają połaczone wartością and. Na przykład: (x_1 or x_2 or not x_3) and ... and (not x_10 or x_n-1 or not x_n) Zostało więc utworzone wyrażenie składające się z takich trójek, o długości wielomianowej.

Mając utworzone takie wyrażenie boolowskie zostanie zaprezentowany algorytm, kóry w efektowny sposób rozwiązuje wyrażenie 3-SAT. Dla argumentów x_1, x_2, ..., x_n jest 2^n możliwych wartościowań. Więc możliwym rozwiązaniem problemu 3-SAT jest sprawdzenie czy przynajmniej jedno wartościowanie spełnia wyrażnie. Jeśli nie ma takiego wartościowania wtedy wyrażenie 3-SAT zawsze zwraca fałsz. Jednak w takim przypadku trzeba sprawdzić wszystkie wartościownania, więc złożonośc takiego algorytmu jest 2^n. Dla n = 40 istnieje 1099511627776 takich możliwości.

W tym artykule zostanie zaprezentowany algorytm, który rozwiązuje ten problem w efektywniejszy sposób.

3. Rozwiązanie

A.

Na początku zostanie stworzona lista wszystkich możliwych trójelementowych elementów ze zbiorów ze wszystkich argumentów. Jest to zbiór ktory zawiera n*(n-1)*(n-2)/6 takich trójek. Każda trójka może mieć 8 wartościowań.
Przykład:

x_1	x_2	x_3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Zbiór prezentuje się więc następująco:

x_1	x_2	x_3	x_4	...	x_n
[0	0	0	-1	...	-1]
[0	0	1	-1	...	-1]
...					

[-1 0 0 0 ... -1]
...
[-1 ... -1 1 1 1]

Finalnie więc zbiór zawiera n*(n-1)*(n-2)/6*8 wartości. Zauważyć można że dla wartości [0 0 0 -1 ... -1] i dla trójki typu (x_1 or x_2 or not x_4) można tą wartość rozszerzyć do [0 0 0 0 -1 .. -1]. Jest to prawidłowe założenie gdyż (x_1 or x_2) dla (0 ,0) wynosi zero a szukane jest prawidłowe rozwiązanie więc mając (x_1 or x_2 or not x_4) i x_1 = 0 oraz x_2 = 0 x_4 musi przyjąć wartość 0 aby (x_1 or x_2 or not x_4) nie odrzucała wartości.

Krokiem w tym algorytmie jest też uzunięcie wartości z listy które dają wartość negatywną dla konkretnej trójki z wyrażenia boolowskiego na przykład dla (x_2 or x_10 or x_n) można usunąć z listy wartośći, które zawierają (x_2 = 0, x_10 = 0, x_n = 0). W algorytmie jest n-2 takich króków gdyż jest n-3 miejsc z wartością -1. Jeśli wszystkie -1 zostaną zastąpione 0 lub 1 to ostanim krokiem jest sprawdzenie czy spełnia ono rozwiązanie. Złożoność każdego podkroku to n*(n-1)*(n-2)/6*8*(n-2)*len(3-SAT) ~ n^4*len(3-SAT). W tym kroku została usunięta część wartości z listy możliwych rozwiązań.

B.

Jednak aby rozwiązać problem 3-SAT potrzebny jest kolejny krok. Dla wartości [0 0 0 -1 ... -1] można znaleźć trójkę która ma jeden wspólny argument. Na przykład trójka (x_2 or x_10 or x_n). Dla tej trójki możliwe wartościowania:

x_2	x_10	x_n
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Dla [0 0 0 -1 ... -1] x_2 = 0. Wiadomo więc że dla x_10 = 0 oraz x_n = 0 nie ma rozwiązania. Więc w tym kroku dodajemy wyrażenie (x_10 or x_n). Dla każdego wartościowania z punktu A tworzone jest osobne wyrażenie, które może więc zawierać zamiast trójek także dwójki. Dla każdej wartości z punktu pierwszego i stworzonego dla niego wyrażenia uruchamiany jest ponownie algorytm z punktu B. Ten algorytm jest uruchamiany dla n-3 argumentów. 3 argumenty są pomijane, gdyż dla nich wartość jest już przypisana.

Każdy krok będzię więc miał złożoność (n-3)*(n-4)*(n-5)/6*8*(n-5)*len(3-SAT) ~ n^4*len(3-SAT). Po wykonaniu algorytmu z punktu A trzeba sprawdzić czy wszystkie możliwe trójki zawierają przynajmniej jedną wartość. Jeśli jest chociaż jedna pusta trójka wartość jest usuwana z listy możliwych rozwiązań dla punktu A.

C.

Teraz mając wartości pozostałe po dwóch poprzednich krokach algorytmu, sprawdzane jest czy kazda trójka zawiera przynajmniej jedną wartość. Jeśli przynajmniej jedna trójka jest bez wartości wiadomo, że problme 3-SAT nie ma rozwiązania, gdyż jeśli istnieje przynajmniej jedno rozwiązanie to każda trójka ma przynajmniej jedną wartość. Aby je znaleźć trzeba najpierw znaleźć wszytkie wartości dla każdej trójki, które znajdują się chociaż w jednym rozwiązaniu. Trzeba więc oczyścić wszystkie wartości pozostałe po dwóch poprzednich krokach algorymtu. Gdyż może się zdarzyć, że dla (x_1, x_2, x_3) jest tylko wartość (0, 0, 0) więc wszystkie trójki które zawierają x_1, x_2 lub x_3 muszą dla tych argumentów mieć wartość 0. Gdy jest 1 to ta wartość jest usuwana.

D.

Teraz mając wszystkie trójki należące do wartościowań spełniających wyrażenie 3-SAT można w efektywny sposób znaleźć jedno wartościowanie

spełniające problem 3-SAT. Znalezenie wszystkich rozwiązań nie jest efektywne gdyż może być na przykład 2^(n-4) wartościowań. Znalezienie jednego rozwiązania jest już możliwe w rozsądnym czasie. Żeby znaleźć to rozwiązanie należy wziąść wartość pierwszą z brzegu. Na przykład wartość dla trójki (x_1, x_2, x_3). Teraz należy wziąć kolejne trójki zawierającej parę (x_1, x_2) która pasuje do pierwszej trójki. Mając częściowe rozwiązanie trzeba sprawdzić wszystkie trójki z argumentów x_1, x_2, x_3, ... , x_k. Przynajmniej jedna wartość z takich trójek musi pasować do częściowego rozwiązania. Jeśli nie jest to spełnione należy wziąść kolejną wartość dla danej trójki x_1, x_2, x_k. Jeśli wszystkie trójki ze zbioru x_1, x_2, ..., x_k mają odpowiednie wartości, wtedy żeby sprawdzić poprawnośc wyrażenie trzeba uruchomić poyższy algorytm dla częściowo obliczonej wartości. Jeśli algorytm nie odrzuci podwartości to znaczy, że częściowa wartość jest poprawna. Znalezienie więć jednego rozwiązania zajmuje:
2*(n-3)*[n*(n-1)*(n-2)/6*8]*[(n-3)*(n-4)*(n-5)/6*8*(n-5)*len(3-SAT)]~
O(n^8*len(3-SAT)) kroków. To jest oczywiście oszacowanie górne. Prawdziwa złożonośc jest niższa. Ale to nie jest cel artykułu.

E.

Należy uzasadnić dlaczego dwa kroki są potrzebne w ukazanym algorytmie. Jest tak gdyż każda wartość może pasować do jednego lub dwóch argumentów z trojki z wyrażenia 3-SAT. Dwa kroki są potrzebne gdyż żeby 3-SAT miało rozwiązanie wszystkie trójki muszą mieć przynajmniej jedną wartość. Jeśli chociaż jedna trójka nie ma żadnych wartości wtedy 3-SAT nie ma rozwiązania. Wykażemy, że dla przynajmniej jednej trójki jest ścieżka która eliminuje wszystkie jej wartości. Poniżej znajduję się lista wszystkich wartości dla wszystkich trójek.

x_1	x_2	x_3	...	x_n
0	0	0		0
0	0	0		1
...				...
1	1	1	...	1

Jeśli dla trójki x_1, x_2, x_3 utworzymy 7 różnych podwyrażeń dla tej trójki usunietych zostanie 7 wartości. Pozostanie więc 2^(n-3) wartości. Następnie gdy usuniemy 7 wartości dla trójki x_4, x_5, x_6 przez utzworzenie 7 różnych podwyrażeń pozostanie 2^(n-6) wartości na liście. Stosując ten krok dla kolejnych trójek zostanie tylko jednatrójka (x_(n-2), x_(n-4), x_(n-3)). Nie usuwaliśmy dla tej trójki żadnej wartości więc pozostanie 8 wartości dla trójki (x_(n-2), x_(n-1), x_n). Będą one następujące:

x_(n-2),	x_(n-1),	x_n
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Jeśli zastosujemy poprzedni krok dla trójki x_(n-2), x_(n-1), x_n zostanie usuniętych 7 wartości. Pozostanie więc jedna tylko wartość. Na przykład:

x_(n-2),	x_(n-1),	x_n
1	1	1

Kandydatem na rozwiązanie będzie więc tylko jedna wartość. Na przykład:

x_1	x_2	x_3	x_4	...	x_(n-3)	x_(n-2)	x_(n-1)	x_n
0	0	0	0	...	0	1	1	1

Dane wyrażenie 3-SAT nie ma rozwiązania, więc żadne wartościowanie nie powinno spełniać tego wyrażenia. Więc tą pozostałą wartość też należy usunąć. Jeśli będzie wyrażenie (not x_(n-2) or not x_(n-1) or not x_n) to ostatnia wartość zostanie usunięta ale wtedy mamy 8 wyrażeń usuwających wszystkie wartości dla danej trójki.

Jeśli zostanie wzięta trójka z dwoma wspólnymi argumentami, np.: (x_1 or

not x_(n-1) or not x_n) to analizując trójkę (x_1, x_(n-1), x_n) mamy wartości

x_1	x_(n-1)	x_n
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

to wtedy stoosując podwyrażenie dla (not x_(n-2) or not x_(n-1) or not x_n) zostanie (0, 1, 1) ale podwyrażenie (x_1 or not x_(n-1) or not x_n) też usunię tą wartość więc stosując algorytm trójka (x_1, x_(n-1), x_n) będzie pustym zbiorem.

Jeśli weźmiemy trójkę z jednym wspólnym argumentem, np.: (x_1 or x_4 or not x_n) wtedy wartość też zostanie usunięta. Dlaczego (x_1 or x_4 or not x_n) też usuwa jakąs trójkę. Można by było sądzić ,że potrzeba trzech kroków algorytmu. Jednak gdy weźmiemy teraz pod uwagę trójkę (x_1, x_2, x_4) to (x_1, x_2) ma dwie wspólne wartości z trójką (x_1, x_2, x_3). Dla dwójki (x_1, x_2) możliwe jest tylko (0, 0). Więc pozostaną dwie wartości dla trójki (x_1, x_2, x_4) mianowice (0, 0, 0) oraz (0, 0, 1). Teraz stosując krok drugi algorytmu dla (0, 0, 1) i mając podwyrażenie z x_4=1 powoduje, że pusta zrobi się trójka (x_4, x_5, x_6). Została więć tylko wartość (x_4=0, x_5=0, x_6=0) a dla x_4 dobra jest tylko 1. (0, 0, 0) też nie pasuje. Stosując algorytm i rozważając trójkę (x_(n-2), x_(n-1), x_n) została dla niej wartość (1, 1, 1). W podwyrażeniu dla wartości (x_1=0, x_2=0, x_4=0) jest x_n=0. Mając podwyrażenie x_n=0 usuwana ona (x_(n_2)=1, x_(n-1)=1, x_n=1). Stąd okazało się, że trójka (x_1, x_2, x_4) po zastosownaiu algorytmu została bez wartości.

Jeśli zrobimy tak dla jakiejkolwiek wartości dla danej trójki z poza argumenów (x_(n-2), x_(n-1), x_n) to stanie się podobnie. Stąd wynik jest taki, że starając się zachować przynajmniej jedną wartość dla danej trójki musimy usunąć wartość dla przynajmniej innej jednej trójki. Pusty zbiór możliwych wartości dla danej trójki oznacza, że istnieje taki zbiór podwyrażeń dla każdej takiej wartości, które usuwają tą wartość z możliwych rozwiązań. To własnie czyni algorytm opisany w tym artykule.

Drugi krok w algorytmie jest więc potrzebny jako, że dla każdego wartościowania dla danej trójki ze zbioru wszystkich możliwych trójek trzeba wygenerować osobne wyrażenie boolowskie podobne do 3-SAT jednak z możliwym dwuargumentowym podwyrażeniem. Oczywiście można uruchomić algorytm pomijając krok pierwszy. I zastosować krok drugi dla wszystkich możliwych wartości ze zbioru wszystkich trójek. Wtedy dla każdej wartości utworzone wyrażenie będzie zawierało podwyrażenie z jednym argumentem. Gdyż musimy tu uwzględnić przypadek gdy dane wartościowanie dla wybranej trójki ma dwa wspólne argumenty z pewnym podwyrażenie z wyrażenia 3-SAT. Jednak zastosowanie kroku pierwszego eliminuje znaczną część wartości dla pewnych trójek. To może ograniczyć działanie kroku drugiego.

Dlaczego nie trzeba użyć 3 lub więcej kroków aby stwierdzić czy wyrażenie 3-SAT ma rozwiązanie lub nie. Np, gdy rozważymy argumenty x_1, x_2, ..., x_7 i weźmiemy podwyrażenia (x_1 or x_2 or x_3), (x_3 or x_4 or x_5) oraz (x_5 or x_6 or x_7) pod podwyrażenie dla pewnej trójki (x_1 or x_2 or x_3) i wartości (1, 1, 0) będzie zawierało dwójkę (x_4 or x_5) ta dwójka może mieć wspólną wartość z (x_5 or x_6 or x_7). Np (x_4=1 x_5=0). Wtedy kolejne podwyrażenie zawiera (x_6 or x_7). Ale ten krok można skrócić rozpatrując od razu wyrażenie (x_3 or x_4 or x_5). Uruchomienie algorytmu dla x_1, x_2, ..., x_7 w takim przypadku i tych trzech wspomnianych podwyrażeń spowoduje, że krok drugi nic nie usunie z żadnej trójki, więc taka zależność nie wnosi nic do algorytmu.

Podsumowując więc złożoność algorytmu wygląda nastęoująco:

1. Pierwszy krok:
   n*(n-1)*(n-2)/6*8*(n-2)*len(3-SAT) ~ O(n^4*len(3-SAT))
2. Drugi krok:
   [n*(n-1)*(n-2)/6*8] * [(n-3)*(n-4)*(n-5)/6*8*(n-5)*len(3-SAT)] ~

O(n^7*len(3-SAT))

2. Finalne czyszczenie:
   [n*(n-1)*(n-2)/6*8*n*(n-1)*(n-2)/6*8*len(3-SAT)] ~ O(n^6*len(3-SAT))
3. Znalezienie jednego rozwiąznia:
   2*(n-3)*[n*(n-1)*(n-2)/6*8]*[(n-3)*(n-4)*(n-5)/6*8*(n-5)*len(3-SAT)]~ O(n^8*len(3-SAT))

Więc maksymalna liczba kroków do znalezienia jednego rozwiązania wynosi n^8*len(3-SAT). Dla 30 argumentów pomijając len(3-SAT) trzeba więc maksymalnie 656100000000 co jest wynikiem gorszym niż 2^30. Lecz dla n=50 2^50=1125899906842624 a 50^7=39062500000000 co jest 28-krotnie lepszym wynikiem. Pominąłem len(3-SAT) gdyż dla prostego algorytmu też powinien on być uwzględniony gdyż dla sprawdzenia konkretnego wartościowania potrzeba sprawdzić len(3-SAT) trójek. Oczywiście te oszacowanie O(n^8*len(3-SAT)) jest ogólne, wszystko zależy od tego co weźmiemy pod uwagę podczas szacownaia złożoności algorytmu. Głównym celem artykułu było wykazanie, że można efektywnie rozwiązać problem 3-SAT. Nawet rozwiązanie o złożoności n^10 jest lepsze niż 2^n. Podsumowując algorytm w czasie wielomianowym stwierdza czy dane wyrażenie 3-SAT nie ma rozwiązania, a jeśli ma to również w czasie wielomianowym zwraca dobre wartościowanie.

4. Podsumowanie

Poniżej załączony został algorytm, który generuje losowe wyrażenie 3-SAT z danej listy argumentów. Kod algorytmu napisany jest w języku programowania python. Język python może służyć jako dobry język programowania do ukazania ogólnego schematu algorytmu. Oczywiście napisanie kodu algoorytmu w innych językach może dawać szybsze rezultaty. W algorytmie poniżej pominięty jest drugi krok, jako że czyszczenie wartości po pierwszym kroku jest tu zastosowane od razu. Algorytm może być modyfikowany i używany w celach komercyjnych ale jego pochodne nie mogą być chronione patentami.

5. Dowód

Całe założenie algorytmu opiera się na tym, że gdy wyrażenie 3-SAT zawsze zwraca fałsz, wtedy algorytm po drugim kroku stwierdza, że istnieje pewna trójka z pustym zbiorem możliwych wartości. Czyli dla pewnej trójki i jej wszystkich wartości istnieje ścieżka która je usuwa.

Dowód twierdzenia operia się na tym, że mając np. 15 argumentow, to wiadomo, że gdy wyrażenie 3-SAT jest fałszywe to żadne podstawienie nie spełnia tego wyrażenia. Biorąc więc trójkę (x_1, x_2, x_3) i wartość (0, 0, 0) można trójki (x_4, x_5, x_6), ..., (x_13, x_14, x_14) przedstawić w formie drzewka o 8 liściach każdy, gdyż jest 8 wartośći dla każdej trójki. Więc dla (x_1, x_2, x_3) jako korzeń drzewa na kolejnym poziomie jest osiem liści z wartościami dla (x_4, x_5, x_6). Dla kolejengo poziomu jest 64 takich liści. Dla ostatniego poziomu 4096. Jeśli dla poziomu drugiego usuniemy 4 wartości to zostanie :

(x_4, x_5, x_6), (x_7 x_8, x_9), (x_10, x_11, x_12), (x_13, x_14, x_15),
 4 8 8 8

gdy usuniemy 4 wartości dla (x_7, x_8, x_9), 4 dla (x_10, x_11, x_12) oraz 4 dla (x_13, x_14, x_15), wtedy:

(x_4, x_5, x_6), (x_7 x_8, x_9), (x_10, x_11, x_12), (x_13, x_14, x_15),
 4 4 4 4

gdy usuniemy 3 dla (x_7, x_8, x_9), 3 dla (x_10, x_11, x_12) oraz 3 dla (x_13, x_14, x_15), wtedy:

(x_4, x_5, x_6), (x_7 x_8, x_9), (x_10, x_11, x_12), (x_13, x_14, x_15),
 4 1 1 1

Zostaną wieć tylko 4 wartości możliwe do spełnienia dla wyrażenia 3-SAT. Zatem dla pewnej trójki musi istnieć ścieżka, która usuwa wszystkie jego 8 wartości. Dla wyjaśnienia jeśli usuniemy 4 wartości dla trójki (x_10, x_11, x_12) z jednej wybranej wartości dla trójki (x_7 x_8, x_9), wtedy dla pozostałych wartości dla trójki (x_7 x_8, x_9) można te 4 wartości też usunąć. Opieramy się tu na

założeniu, że sekwencja podwyrażeń dla 3-SAT typu (x_1 or x_2 or x_3), (x_3 or x_4 or x_5) oraz (x_5 or x_6 or x_7) nic nie wnosi do wyników, stąd można od razu rozpatrywać ścieżkę od (x_3 or x_4 or x_5) do (x_5 or x_6 or x_7).

6. Spostrzeżenia

Po perwszym uruchomieniu kroku A, może się okazać, że tymczasowe wartości zawierają jedną wartość bez -1. Wtedy taką wartość można wziąść od razu i zwrócić jako wynik algorytmu.

Bibliogriafa:

1. "Wprowadzenie do algorytmów", T. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, 2022
2. Wikipedia
3. www.jetbrains.com – Oprogramowanie PyCharm służące tworzeniu implementacji algorytmu, oraz testowania jego poprawności.

1. Algorithm:
    1. Function: Steps from point A in article:
        1. Generate all values for all threes.
        2. For every value check if subexpression exists that should remove it
        3. For n=1,..,len(list_of_args)-1
            1. For every value do:
                1.Check if value can be extended for any subexpression
                2. Remove extended value if subexpression exists that should remove it

        4 .Check if all threes have at least one value. If no return false, else return list of remaining values.

1. Input
    1. list_of_args
    2. list_of_3sat_subexpressions
2. For input value apply function above.
3. For every remaining value create list of subexpression with two args possible.
4. For every remaining value apply function again.
5. Do cleaning
6. Check if all threes have at least one value. If no return false.
7. Having all good remaining values for all threes do:
    1. Take first value for any three.
    2. For every next three with thwo common args:
        A. For next possible value
            1. Check if partial value is good with all good remianing values
            2. If is good create subexpression for given subvalue and apply function above
            3. If it is true then if all values have 0 or 1 return value else do step A again. If it is false got to ponit A.

Algorithm in python

```
import random
import copy
import math


def generate_all_values(args_dict):
    args_dict_keys = list(args_dict.keys())
    args_dict_keys.sort()
    if len(args_dict) == 0:
        return [[]]
    tmp_values = [0] * len(args_dict_keys)
    for x in range(0, 2**len(args_dict)):
        tmp_values = copy.deepcopy(tmp_values)
        if x == 0:
            yield tmp_values
        for n in range(0, len(args_dict)):
            if tmp_values[n] == 0:
                tmp_values[n] = 1
                for t in range(0, n):
                    tmp_values[t] = 0
                break
        yield tmp_values


def test_3sat_real(clause_number, args_dict):
    global good_assignments
    good_assignments = []
    string_clause = ""
    for clause in clause_number:
        tmp_string_clause = ""
        for counter in range(len(clause[1])):
            if clause[0][counter]:
                tmp_string_clause += clause[1][counter] + "+"
            else:
                tmp_string_clause += "!" + clause[1][counter] + "+"
        string_clause += "(" + tmp_string_clause[:-1] + ")^"
    print(string_clause[:-1])
    args_dict_keys = list(args_dict.keys())
    args_dict_keys.sort()
    good_assignments = []
    dividend = math.pow(2, len(args_dict) - 5)
    index = 0
    args_dict_tmp = copy.deepcopy(args_dict)
    for tmp_values in generate_all_values(args_dict_tmp):
        for arg in args_dict_keys:
            if arg not in args_dict_tmp:
                tmp_values.insert(args_dict_keys.index(arg), 0)
        final_result = 1
        final_result_string = ""
        for clause in clause_number:
            tmp_final_result = calculate_real_clause_list(clause,
args_dict_keys, tmp_values)
```

```
            if tmp_final_result == 0:
                break
            final_result_string += str(tmp_final_result) + "^"
            final_result &= tmp_final_result
        if final_result == 1 and tmp_final_result == 1:
            good_assignments.append(tmp_values)
        index += 1
        if index % dividend == 0:
            print(f"index {index}")
    print("Number goods: " + str(len(good_assignments)))
    return good_assignments


def calculate_real_clause_list(clause, args_keys, args_values):
    result = 0
    for counter in range(0, len(clause[0])):
        index = args_keys.index(clause[1][counter])
        arg = args_values[index]
        if not clause[0][counter]:
            if arg:
                arg = 0
            else:
                arg = 1
        result |= arg
    return result


def generate_real_clause(*args_list):
    args_number = 3 #random.randint(1, len(args_list))
    args_clause = [0] * args_number
    args_litteral = [0] * args_number
    for count  in range(args_number):
        args_clause[count] = random.randint(0, 1)
        argument = args_list[random.randint(0, len(args_list) - 1)]
        while argument in args_litteral:
            argument = args_list[random.randint(0, len(args_list) - 1)]
        args_litteral[count] = argument
    tmp_args_litteral = args_litteral.copy()
    tmp_args_litteral.sort()
    tmp_args_clause = [0] * args_number
    for counter in range(0, len(tmp_args_litteral)):
        tmp_args_clause[counter] =
args_clause[args_litteral.index(tmp_args_litteral[counter])]
    return tmp_args_clause, tmp_args_litteral


def gen_3sat_clause(args_dict, clause_n):
    clause_number = [[]] * clause_n
    for count in range(len(clause_number)):
        tmp_clause = generate_real_clause(*args_dict)
        while tmp_clause in clause_number:
            tmp_clause = generate_real_clause(*args_dict)
        clause_number[count] = tmp_clause
    string_clause = ""
    for clause in clause_number:
        tmp_string_clause = ""
        for counter in range(len(clause[1])):
            if clause[0][counter]:
                tmp_string_clause += clause[1][counter] + "+"
            else:
                tmp_string_clause += "!" + clause[1][counter] + "+"
```

```
        string_clause += "(" + tmp_string_clause[:-1] + ")^"
    print(string_clause[:-1])
    print("=============================================")
    return clause_number


def find_solution(clause_number, args_dict, number_real, quick=False):
    print("Clause:", clause_number)
    string_clause = ""
    for clause in clause_number:
        tmp_string_clause = ""
        for counter in range(len(clause[1])):
            if clause[0][counter]:
                tmp_string_clause += clause[1][counter] + "+"
            else:
                tmp_string_clause += "!" + clause[1][counter] + "+"
        string_clause += "(" + tmp_string_clause[:-1] + ")^"
    print("Clause", string_clause)
    assignment_excluded = []
    for clause in clause_number:
        tmp_assignment = {}
        for counter in range(len(clause[1])):
            if clause[0][counter]:
                tmp_assignment[clause[1][counter]] = 0
            else:
                tmp_assignment[clause[1][counter]] = 1
        assignment_excluded.append(tmp_assignment)
    good_list_tmp = {}
    for x in args_dict.keys():
        for y in args_dict.keys():
            for z in args_dict.keys():
                if x != y and x != z and y != z:
                    key = [x, y, z]
                    key.sort()
                    if tuple(key) not in good_list_tmp.keys():
                        good_list_tmp[tuple(key)] = [[0, 0, 0], [0, 0, 1], [0,
1, 0], [0, 1, 1],
                                                     [1, 0, 0], [1, 0, 1], [1,
1, 0], [1, 1, 1]]
    full_key = list(args_dict.keys())
    full_key.sort()
    print("Arguments", full_key)
    good_list = []
    good_list_debug = []
    full_three = list(good_list_tmp.keys())
    for g_key in good_list_tmp:
        for t_val in good_list_tmp[g_key]:
            tmp_val = []
            for f_key in full_key:
                if f_key in g_key:
                    tmp_val.append(t_val[g_key.index(f_key)])
                else:
                    tmp_val.append(-1)
            good_list.append(tmp_val)
            good_list_debug.append([t_val, g_key])
    expression_list = []
    for assignment in assignment_excluded:
        keys = list(assignment.keys())
        keys.sort()
        values = []
        for key in keys:
```

```
            values.append(assignment[key])
        expression_list.append((tuple(keys), values))
    assignment_found = []
    good_to_remove = []
    print("Length of list before exclude", len(good_list))
    print("Expresion list", expression_list)
    for good_value in good_list:
        for keys, values, in expression_list:
            for index_key, expression_key in enumerate(keys):
                if good_value[full_key.index(expression_key)] !=
values[index_key]:
                    break
            else:
                if keys not in assignment_found:
                    assignment_found.append(keys)
                if good_value not in good_to_remove:
                    good_to_remove.append(good_value)
    for good in good_to_remove:
        good_list_debug.pop(good_list.index(good))
        good_list.remove(good)
    print("Length of list after exclude", len(good_list))
    good_list_not = []
    good_list_debug_not = []
    for step in range(0, len(full_key) - 1):
        print(step)
        was_changed = False
        goods_to_remove = []
        good_list_not_add = []
        for good_value in good_list:
            for expression_keys, expression_value in expression_list:
                vals_found = 0
                key_not_found = None
                for index_key, expression_key in enumerate(expression_keys):
                    if good_value[full_key.index(expression_key)] ==
expression_value[index_key]:
                        vals_found += 1
                    elif good_value[full_key.index(expression_key)] == -1:
                        key_not_found = expression_key
                if vals_found == (len(expression_keys) - 1) and key_not_found:
                    if expression_value[expression_keys.index(key_not_found)] ==
0:
                        good_value[full_key.index(key_not_found)] = 1
                    else:
                        good_value[full_key.index(key_not_found)] = 0
                elif vals_found == len(expression_keys):
                    goods_to_remove.append(good_value)
                else:
                    continue
                was_changed = True
                break
            else:
                good_list_not_add.append(good_value)
        for good_to_add in good_list_not_add:

good_list_debug_not.append(good_list_debug[good_list.index(good_to_add)])
            good_list_debug.pop(good_list.index(good_to_add))
            good_list_not.append(good_to_add)
            good_list.remove(good_to_add)
        for good_to_remove in goods_to_remove:
            good_list_debug.pop(good_list.index(good_to_remove))
            good_list.remove(good_to_remove)
```

```
        if not was_changed:
            break
    for indx, good in enumerate(good_list_not):
        good_list.append(good)
        good_list_debug.append(good_list_debug_not[indx])
    print("len after", len(good_list))
    threes_analyze = {}
    for val, key in good_list_debug:
        # if len(val) < 8:
        #     print(key, val)
        for three in full_three:
            for three_s in three:
                if three_s not in key:
                    break
            else:
                if three not in threes_analyze:
                    threes_analyze[three] = []
                if val not in threes_analyze[three]:
                    threes_analyze[three].append(val)
    for key in full_three:
        if key not in threes_analyze:
            print("Key is missing. No solution", key)
            return False
    three_to_three_double = {}
    three_to_three_one = {}
    for three_1 in threes_analyze:
        three_to_three_double[three_1] = []
        three_to_three_one[three_1] = []
        for three_2 in threes_analyze:
            counter = 0
            three_in = []
            three_not_in = []
            for indx, three_2_s in enumerate(three_2):
                if three_2_s in three_1:
                    counter += 1
                    three_in.append([three_2_s, indx])
                else:
                    three_not_in.append([three_2_s, indx])
            if counter == 2:
                three_to_three_double[three_1].append([three_2, three_in,
three_not_in])
            if counter == 1:
                three_to_three_one[three_1].append([three_2, three_in,
three_not_in])
    for i in range(0, len(full_key)):
        print("Cleaning step", i)
        was_not_in = False
        counter_clean = 0
        for three in threes_analyze:
            val_to_remove = []
            for val in threes_analyze[three]:
                double_list = {}
                one_list = {}
                was_removed = False
                for three_c, three_in, three_not_in in
three_to_three_one[three]:
                    double_key = (three_not_in[0][0], three_not_in[1][0])
                    double_tmp = []
                    was_in = False
                    for val_c in threes_analyze[three_c]:
                        if val_c[three_in[0][1]] == val[three.index(three_in[0]
```

```
[0])]:
                                double_tmp.append([val_c[three_not_in[0][1]],
val_c[three_not_in[1][1]]])
                                was_in = True
                        if not was_in:
                            if val not in val_to_remove:
                                val_to_remove.append(val)
                                was_removed = True
                        if double_key not in double_list:
                            double_list[double_key] = double_tmp
                        else:
                            to_rem = []
                            for double_val in double_list[double_key]:
                                if double_val not in double_tmp:
                                    to_rem.append(double_val)
                            for rem in to_rem:
                                double_list[double_key].remove(rem)
                            if not double_list[double_key]:
                                if val not in val_to_remove:
                                    val_to_remove.append(val)
                                    was_removed = True
                        if was_removed:
                            break
                    if was_removed:
                        break
                    for three_c, three_in, three_not_in in
three_to_three_double[three]:
                        one_key = (three_not_in[0][0])
                        one_tmp = []
                        was_in = False
                        for val_c in threes_analyze[three_c]:
                            if (val_c[three_in[0][1]] == val[three.index(three_in[0]
[0])]
                                    and val_c[three_in[1][1]] ==
val[three.index(three_in[1][0])]):
                                was_in = True
                                one_tmp.append(val_c[three_not_in[0][1]])
                        if not was_in:
                            if val not in val_to_remove:
                                val_to_remove.append(val)
                                was_removed = True
                        if one_key not in one_list:
                            one_list[one_key] = one_tmp
                        else:
                            to_rem = []
                            for one_val in one_list[one_key]:
                                if one_val not in one_tmp:
                                    to_rem.append(one_val)
                            for rem in to_rem:
                                one_list[one_key].remove(rem)
                            if not one_list[one_key]:
                                if val not in val_to_remove:
                                    val_to_remove.append(val)
                                    was_removed = True
                        if was_removed:
                            break
                    tmp_keys = {}
                    for double_key in double_list:
                        one_tmp_1 = []
                        one_tmp_2 = []
                        for val_l in double_list[double_key]:
```

```
                        if val_l[0] not in one_tmp_1:
                            one_tmp_1.append(val_l[0])
                        if val_l[1] not in one_tmp_2:
                            one_tmp_2.append(val_l[1])
                    if double_key[0] not in tmp_keys:
                        tmp_keys[double_key[0]] = one_tmp_1
                    else:
                        to_rem = []
                        for one_t in tmp_keys[double_key[0]]:
                            if one_t not in one_tmp_1:
                                to_rem.append(one_t)
                        for rem in to_rem:
                            tmp_keys[double_key[0]].remove(rem)
                    if double_key[1] not in tmp_keys:
                        tmp_keys[double_key[1]] = one_tmp_2
                    else:
                        to_rem = []
                        for one_t in tmp_keys[double_key[1]]:
                            if one_t not in one_tmp_2:
                                to_rem.append(one_t)
                        for rem in to_rem:
                            tmp_keys[double_key[1]].remove(rem)
                for tmp in tmp_keys:
                    if not tmp_keys[tmp]:
                        if val not in val_to_remove:
                            val_to_remove.append(val)
            for val in val_to_remove:
                threes_analyze[three].remove(val)
                if len(threes_analyze[three]) == 0:
                    print("Missing key. No solution", three)
                    return False
                counter_clean += 1
                was_not_in = True
        print("Counter clean", counter_clean)
        if not was_not_in:
            print("was not in")
            break
    if quick:
        return True
    print("Good three list clean")
    for three in threes_analyze:
        print(three, threes_analyze[three])
    print("----")
    print("------------------- three vals after excluding")
    final_good_first = [-1] * len(full_key)
    first_three = (full_key[0], full_key[1], full_key[2])
    first_three_val = threes_analyze[first_three][0]
    final_good_first[0] = first_three_val[0]
    final_good_first[1] = first_three_val[1]
    final_good_first[2] = first_three_val[2]
    final_good = find_final(final_good_first, first_three, threes_analyze,
                            full_key, assignment_excluded, clause_number)
    print("Full key", full_key)
    print("Final good:", final_good)
    final_result = 1
    for clause in clause_number:
        tmp_final_result = calculate_real_clause_list(clause, full_key,
final_good)
        print("Clause", clause, tmp_final_result)
        final_result &= tmp_final_result
    if final_result == 0:
```

```
        print("Final value does not solve the 3-SAT expression but should")
        exit(1)
    return True


def check_if_good(tmp_final_good, full_key, assignment_excluded):
    min_values_minus_excluded = []
    for ass_exc in assignment_excluded:
        not_good = {}
        for ass in ass_exc:
            if tmp_final_good[full_key.index(ass)] == -1:
                if ass_exc[ass] == 0:
                    not_good[ass] = 1
                else:
                    not_good[ass] = 0
                #not_good[ass] = ass_exc[ass]
            elif ass_exc[ass] != tmp_final_good[full_key.index(ass)]:
                break
        else:
            #print(ass_exc)
            if not_good and not_good not in min_values_minus_excluded:
                min_values_minus_excluded.append(not_good)
    clauses = []
    for min_values_minus in min_values_minus_excluded:
        clause = ([], [])
        for min in min_values_minus:
            clause[1].append(min)
        clause[1].sort()
        for cl in clause[1]:
            clause[0].append(min_values_minus[cl])
        clauses.append(clause)
    args_dict = {}
    for indx, tmp_val in enumerate(tmp_final_good):
        if tmp_val == -1:
            args_dict[full_key[indx]] = 0
    print(min_values_minus_excluded)
    return find_solution(clauses, args_dict, [], quick=True)


def check_tmp_final(tmp_final_good, threes_analyze, full_key):
    for three_c in threes_analyze:
        three_in = []
        for three_c_s in three_c:
            if tmp_final_good[full_key.index(three_c_s)] != -1:
                three_in.append(three_c_s)
        if three_in:
            is_good = False
            for val_c in threes_analyze[three_c]:
                for three_c_s in three_in:
                    if val_c[three_c.index(three_c_s)] !=
tmp_final_good[full_key.index(three_c_s)]:
                        break
                else:
                    is_good = True
            if not is_good:
                print("is good false", three_c, threes_analyze[three_c])
                return False
    return True


def find_final(tmp_final_good, first_three, threes_analyze, full_key,
```

```
assignment_excluded, clause_number):
    print("First three", first_three)
    for key in full_key[3:]:
        current_three = (first_three[0], first_three[1], key)
        if key == full_key[-3]:
            for last_vals in [[0, 0, 0], [0, 0, 1], [0, 1, 0], [0, 1, 1],
                              [1, 0, 0], [1, 0, 1], [1, 1, 0], [1, 1, 1]]:
                tmp_final_last = copy.deepcopy(tmp_final_good)
                tmp_final_last[-3] = last_vals[0]
                tmp_final_last[-2] = last_vals[1]
                tmp_final_last[-1] = last_vals[2]
                final_result = 1
                for clause in clause_number:
                    tmp_final_result = calculate_real_clause_list(clause,
full_key, tmp_final_last)
                    # print("Clause", clause, tmp_final_result)
                    final_result &= tmp_final_result
                if final_result == 0:
                    print("Final candidate value does not solve the 3-SAT
expression")
                else:
                    return tmp_final_last
            else:
                print("No assignment")
                exit(1)
        candidates = []
        for val in threes_analyze[current_three]:
            if val[0] == tmp_final_good[0] and val[1] == tmp_final_good[1]:
                candidates.append(val[2])
        print(candidates)
        if len(candidates) == 1:
            tmp_final_good[full_key.index(key)] = candidates[0]
        else:
            for candidate in candidates:
                tmp_final = copy.deepcopy(tmp_final_good)
                print("Run for ", tmp_final, current_three, current_three,
candidate)
                tmp_final[full_key.index(key)] = candidate
                if check_tmp_final(tmp_final, threes_analyze, full_key):
                    if check_if_good(tmp_final, full_key, assignment_excluded):
                        tmp_final_good = tmp_final
                        print("Is good", tmp_final)
                        break
                print("Is not good", tmp_final)
            else:
                print("No solution")
                exit(1)
    return tmp_final_good


def main():
    args_dict = {}
    #for var in ["a", "x"]:
    #for var in ["a", "b", "c"]:+
    #for var in ["a", "x", "y", "u"]:
    #for var in ["a", "b", "c", "d", "e"]:
    #for var in ["a", "b", "c", "d", "e", "f"]:
    #for var in ["a", "b", "c", "d", "e", "f", "g", "h"]:
    #for var in ['a', 'b', 'c', 'd', 'e', 'u', 'w', 'x']:
    for var in ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']:
    #for var in ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l']:
```

```
    #for var in ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'u', 'w', 'x', 'y', 'z']:
    #for var in ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l',
'm']:
    #for var in ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l',
'm', 'n', 'o']:
        #for i in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
        #for i in [1, 2, 3, 4, 5, 6, 7, 8]:
        #for i in [1, 2, 3, 4, 5, 6]:
        for i in [1, 2, 3, 4]:
        #for i in [1, 2, 3]:
        #for i in [1, 2]:
        #for i in [1]:
            args_dict["%s%s" % (var, i)] = 0
    expression = gen_3sat_clause(args_dict, 105)  # 45)
    print("Expression", expression)
    #number_real = test_3sat_real(expression, args_dict)
    number_real = []
    #print("Number real: ", len(number_real), str(number_real[:10]))
    find_solution(expression, args_dict, number_real)


if __name__ == "__main__":
    main()
```