Полиномиален алгоритъм, основан на метода на резолюцията, за частен случай на задачата 3-SAT

Траян Танчев Господинов
(студент от ФМИ на СУ "Св. Климент Охридски", № 45506)

19 декември 2021 г.

Резюме

Ще изложим алгоритъм, който решава голяма част от екземплярите на задачата 3-SAT за време $O(n^4)$ и с памет $\Theta(n^3)$, където n е броят на съждителните променливи, участващи във входната формула на 3-SAT. Алгоритъмът връща един от следните три отговора:

1) "няма решение", ако алгоритъмът е сигурен, че входната формула е неудовлетворима;
2) наредена n-торка от стойностите на съждителните променливи, ако алгоритъмът е открил решение;
3) "неизвестно", ако алгоритъмът не е сигурен за отговора.

В първите два случая резултатът от алгоритъма със сигурност е верен. Третият случай се среща много рядко.

Теоретични положения

Дизюнкт се нарича крайно множество от литерали. Като всяко множество дизюнктът притежава мощност (брой елементи). Ще означаваме празния дизюнкт с ■. N-дизюнкт ще наричаме дизюнкт D, за който $|D| \le N$.

Нека D_1 и D_2 са N-дизюнкти, а L е литерал. Ще казваме, че правилото за съждителна N-резолюция е приложимо към D_1 и D_2 относно L, ако $L \in D_1$, $L^\partial \in D_2$ и $\left|(D_1 \setminus \{L\}) \cup (D_2 \setminus \{L^\partial\})\right| \le N$. Означение: $\mathcal{R}_L^N(D_1, D_2)$. Правилото за съждителна N-резолюция поражда N-дизюнкт: $\mathcal{R}_L^N(D_1, D_2) \stackrel{\text{def}}{=} (D_1 \setminus \{L\}) \cup (D_2 \setminus \{L^\partial\})$.

Нека D, D_1 и D_2 са N-дизюнкти. Ще казваме, че D е N-резолвента на D_1 и D_2, ако съществува литерал L, за който $D = \mathcal{R}_L^N(D_1, D_2)$.

За произволно множество S, съставено от N-дизюнкти, определяме една операция R_N по следния начин: $R_N(S) \stackrel{\text{def}}{=} S \cup \{$ N-резолвентите на всевъзможните двойки дизюнкти от $S\}$.

Лема 1. Нека D е N-резолвента на D_1 и D_2, а I е булева интерпретация. Ако $I \models \{D_1, D_2\}$, то $I \models \{D\}$.

Лема 2. Нека S е множество от N-дизюнкти, а I е булева интерпретация. Тогава $I \models S \Leftrightarrow I \models R_N(S)$.

Доказателство: Необходимостта следва от лема 1. Достатъчността следва от това, че $S \subseteq R_N(S)$.

Нека S е множество от N-дизюнкти. Рекурентно определяме една редица от множества от N-дизюнкти: $S_0 = S$, $S_{n+1} = R_N(S_n)$. Тогава $S_N^* \stackrel{\text{def}}{=} \bigcup_{i=0}^{\infty} S_i$ също е множество от N-дизюнкти и се нарича насищане на S. От определението на операцията R_N следва непосредствено, че $S_0 \subseteq S_1 \subseteq S_2 \subseteq S_3 \subseteq \ldots$

Лема 3. Нека S е множество от N-дизюнкти, а I е булева интерпретация. Тогава $I \models S \Leftrightarrow I \models S_N^*$.

Доказателство: Достатъчността следва от включването $S \subseteq S_N^*$. Необходимостта следва от твърдението $I \models S_n$, което на свой ред следва от лема 2 чрез математическа индукция по n.

За едно множество S от N-дизюнкти казваме, че е удовлетворимо, ако съществува булева интерпретация I, за която $I \models S$ (тоест всеки дизюнкт на S съдържа поне един литерал със стойност *истина* относно I).

Лема 4. Нека S е множество от N-дизюнкти. Ако $\blacksquare \in S_N^*$, то множеството S е неудовлетворимо.

Доказателство: Да допуснем: че $I \models S$ за някоя булева интерпретация I. От лема 3 следва, че $I \models S_N^*$. Понеже $\blacksquare \in S_N^*$, то празният дизюнкт ■ съдържа литерал със стойност *истина* относно I, което е противоречие.

Нека S е множество от N-дизюнкти. Казваме, че S е затворено относно операцията R_N, ако $R_N(S) = S$, тоест ако S съдържа N-резолвентите на всеки два свои дизюнкта.

Твърдение 1. За всяко множество S от N-дизюнкти насищането S_N^* е затворено относно операцията R_N.

Доказателство: Трябва да докажем, че $R_N\left(S_N^*\right) = S_N^*$. Ще докажем двете включвания поотделно. Включването $R_N\left(S_N^*\right) \supseteq S_N^*$ следва пряко от определението на операцията R_N. Включването $R_N\left(S_N^*\right) \subseteq S_N^*$ следва от това, че множеството S_N^* съдържа N-резолвентите на всеки два свои N-дизюнкта. Действително, нека D_1 и D_2 са N-дизюнкти от S_N^* и D е N-резолвента на D_1 и D_2. Тогава съществуват цели неотрицателни числа i и j, за които $D_1 \in S_i$ и $D_2 \in S_j$ (където S_i и S_j са членове на редицата от определението на насищане). Без ограничение можем да предположим, че $i \geq j$. Тогава $S_i \supseteq S_j$, поради което S_i съдържа дизюнктите D_1 и D_2. Следователно $S_{i+1} = R_N\left(S_i\right)$ съдържа тяхната N-резолвента D. Тъй като $S_N^* \supseteq S_{i+1}$, то S_N^* съдържа D, което трябваше да се докаже.

Нека S е множество от N-дизюнкти. N-резолютивен извод от S наричаме крайна редица от N-дизюнкти, всеки член на която е от S или е N-резолвента на някои два предходни члена на редицата. За един N-дизюнкт D казваме, че е N-резолютивно изводим от S, и пишем $S \overset{r}{\underset{N}{\vdash}} D$, ако съществува N-резолютивен извод от S с последен член D.

Твърдение 2. Нека D е N-дизюнкт, а S е множество от N-дизюнкти. Тогава $D \in S_N^* \Leftrightarrow S \overset{r}{\underset{N}{\vdash}} D$.

Доказателство: *Необходимост:* Щом $D \in S_N^*$, то съществува цяло неотрицателно n, за което $D \in S_n$. Оттук с индукция по n следва, че $S \overset{r}{\underset{N}{\vdash}} D$. База: $n = 0$; тогава $D \in S$ и N-резолютивният извод съдържа само D. При $n > 0$: индуктивна стъпка от $n - 1$ към n: ако $D \in S_{n-1}$, прилагаме индуктивното предположение към D; иначе D е N-резолвента на някакви D_1 и D_2 от S_{n-1}, към които прилагаме индуктивното предположение, а после слепваме двата N-резолютивни извода и добавяме D в края на получената редица от N-дизюнкти.

Достатъчност: Следва от твърдение 1 чрез индукция по дължината на N-резолютивния извод.

От лема 4 и твърдение 2 получаваме следната **теорема за коректност на *N*-резолютивната изводимост:** Нека S е множество от N-дизюнкти. Ако $S \overset{r}{\underset{N}{\vdash}} \blacksquare$, то множеството S е неудовлетворимо.

Импликациите в тази теорема и в лема 4 са еднопосочни. При $N > 2$ обратните импликации не са верни, тоест N-резолютивната изводимост не е пълна.

Очевидна е връзката на разгледаните понятия с алгоритмичната задача SAT:
- множество от дизюнкти съответства на конюнктивна нормална форма;
- всеки отделен дизюнкт съответства на една клауза;
- изводимостта на празния дизюнкт е равносилна на изводимостта на противоречиво следствие, поради което влече неудовлетворимост на конюнктивната нормална форма.

Известно е, че общият случай на задачата SAT се свежда до частния случай 3-SAT, при който всяка клауза е дизюнкция с точно три операнда, всеки от които е отделна логическа променлива със или без отрицание. Това почти съответства на 3-дизюнкт, само че в определението на 3-дизюнкт се допуска променливите да бъдат по-малко от три. Неточното съответствие не е пречка, защото можем да разгледаме този леко обобщен вариант на алгоритмичната задача 3-SAT, в който разрешаваме клаузите на формулата да съдържат три или по-малко съждителни променливи.

И тъй, що се отнася до понятията N-дизюнкт, N-резолвента и др., оттук нататък ще се съсредоточим върху частния случай $N = 3$.

Следствие. Нека S е множество от 3-дизюнкти. Ако $S \overset{r}{\underset{3}{\vdash}} \blacksquare$, то множеството S е неудовлетворимо.

Това твърдение се получава непосредствено от теоремата за коректност на N-резолютивната изводимост чрез заместването $N = 3$.

Импликацията в следствието е еднопосочна. Обратната импликация не е вярна.

Полученото следствие стои в основата на алгоритъм за задачата 3-SAT.

Идейно описание на алгоритъм за задачата 3-SAT

Входът на алгоритъма е конюнктивна нормална форма, разглеждана като множество S от 3-дизюнкти. Нека n е броят на съждителните променливи.

1) $S' \leftarrow S_3^*$.
2) Ако $\blacksquare \in S'$, тоест $\boldsymbol{S} \overset{r}{\underset{3}{\vdash}} \blacksquare$, то алгоритъмът връща отговор "няма решение" и приключва работа (конюнктивната нормална форма е неудовлетворима).
3) Ако S' съдържа или $\{A\}$, или $\{\neg A\}$ за всяка съждителна променлива A, то следва стъпка 8.
4) Към S' се добавя $\{A\}$, където A е някоя съждителна променлива, за която нито $\{A\}$, нито $\{\neg A\}$ не се съдържа в S'.
5) $S' \leftarrow (S')_3^*$.
6) Ако $\blacksquare \in S'$, то алгоритъмът връща отговор "неизвестно" и приключва работа.
7) Алгоритъмът отива на стъпка 3.
8) На всяка съждителна променлива A се присвоява стойност по следното правило:
   ако S' съдържа $\{A\}$, то на A се присвоява стойност *истина*;
   ако S' съдържа $\{\neg A\}$, то на A се присвоява стойност *лъжа*.
9) Изчислява се стойността на формулата. Ако тя е *истина*, алгоритъмът връща намереното решение (конюнктивната нормална форма е удовлетворима); иначе алгоритъмът връща отговор "неизвестно".

Коректност на алгоритъма:

— Отговор "няма решение" се връща само от стъпка 2 и е правилен съгласно със следствието за $N = 3$ от теоремата за коректност на N-резолютивната изводимост.

— Конкретно решение (редица от стойности на съждителните променливи) се връща само от стъпка 9, и то винаги след проверка (чрез заместване в дадената логическа формула). Тази проверка дава гаранция, че намереното решение е правилно.

— Във всички останали случаи алгоритъмът връща отговор "неизвестно"; тук няма какво да се доказва.

На този етап не можем да анализираме сложността на алгоритъма по време и памет, тъй като все още не сме уточнили използваните структури от данни и отделни действия, например как се изчислява насищането. Все пак нека отбележим, че стъпка 4 е ключът към бързодействието на алгоритъма: изпробва се само едната от двете възможни логически стойности на променливата A, с което се избягва комбинаторният взрив.

Подробно описание на алгоритъма във вид на псевдокод

За ефективна реализация на алгоритъма ще ни трябват следните девет правила:

1. $(A) \wedge (\neg A) \rightarrow ()$.
2. $(A) \wedge (\neg A \vee B) \rightarrow (B)$.
3. $(A) \wedge (\neg A \vee B \vee C) \rightarrow (B \vee C)$.
4. $(A \vee B) \wedge (\neg A) \rightarrow (B)$.
5. $(A \vee B) \wedge (\neg A \vee C) \rightarrow (B \vee C)$.
6. $(A \vee B) \wedge (\neg A \vee C \vee D) \rightarrow (B \vee C \vee D)$.
7. $(A \vee B \vee C) \wedge (\neg A) \rightarrow (B \vee C)$.
8. $(A \vee B \vee C) \wedge (\neg A \vee D) \rightarrow (B \vee C \vee D)$.
9. $(A \vee B \vee C) \wedge (\neg A \vee B \vee D) \rightarrow (B \vee C \vee D)$.

Тези правила изчерпват възможностите на 3-резолюцията.

В глобална променлива n се пази броят на съждителните променливи на конюнктивната нормална форма. Един глобален едномерен логически масив disjuncts с дължина $1 + 2n + (2n)^2 + (2n)^3$ съдържа множеството на генерираните 3-дизюнкти, включително тавтологиите:

$$\blacksquare, \{A\}, \{\neg A\}, \ \ldots\ , \{A, B\}, \{A, \neg B\}, \{\neg A, B\}, \{\neg A, \neg B\}, \{A, \neg A\}, \ \ldots\ , \{A, B, C\}, \ldots$$

Функцията getCode(disjunct) по даден 3-дизюнкт определя неговия индекс в глобалния масив disjuncts; следователно тази функция връща цяло число от 0 до disjuncts.size() – 1 включително.

Алгоритъмът използва и една глобална опашка q от генерирани, но необработени 3-дизюнкти.

Функция за записване на генерираните 3-дизюнкти в опашката и в логическия масив:

```
1.addDisjunct(disjunct):
2.      if disjuncts[getCode(disjunct)]=FALSE then
3.              disjuncts[getCode(disjunct)]←TRUE
4.              q.push(disjunct)
```

Правилата на 3-резолюцията:

```
//(A) (¬ A) → ()
1. resolution11(disjunct): //disjunct = {A}
2.      if disjuncts[getCode(-A)]=TRUE then
3.              addDisjunct(makeEmptyDisjunct())

//(A) (¬ A ∨ B) → (B)
1. resolution12(disjunct): //disjunct = {A}
2.      for B∈Literals:
3.              if disjuncts[getCode(makeDisjunct(-A, B))]=TRUE then
4.                      addDisjuncts(makeDisjunct(B))

//(A) (¬ A ∨ B ∨ C) → (B ∨ C)
1. resolution13(disjunct): //disjunct = {A}
2.      for B,C∈Literals:
3.              if disjuncts[getCode(makeDisjuncts(-A, B, C))]=TRUE then
4.                      addDisjunct(makeDisjunct(B, C))
```

Аналогично се реализират и останалите шест правила:
resolution21(disjunct); // $(A \vee B) \wedge (\neg A) \rightarrow (B)$;
resolution22(disjunct); // $(A \vee B) \wedge (\neg A \vee C) \rightarrow (B \vee C)$;
resolution23(disjunct); // $(A \vee B) \wedge (\neg A \vee C \vee D) \rightarrow (B \vee C \vee D)$;
resolution31(disjunct); // $(A \vee B \vee C) \wedge (\neg A) \rightarrow (B \vee C)$;
resolution32(disjunct); // $(A \vee B \vee C) \wedge (\neg A \vee D) \rightarrow (B \vee C \vee D)$;
resolution33(disjunct); // $(A \vee B \vee C) \wedge (\neg A \vee B \vee D) \rightarrow (B \vee C \vee D)$.

Върху всеки непразен 3-дизюнкт се прилагат три от деветте правила; избират се според мощността на 3-дизюнкта:

```
1. resolution(disjunct):
2.      if disjunct.size()=0 then
3.              return
4.      if disjunct.size()=1 then
5.              resolution11(disjunct)
6.              resolution12(disjunct)
7.              resolution13(disjunct)
8.      if disjunct.size()=2 then
9.              resolution21(disjunct)
10.             resolution22(disjunct)
11.             resolution23(disjunct)
12.     if disjunct.size()=3 then
13.             resolution31(disjunct)
14.             resolution32(disjunct)
15.             resolution33(disjunct)
```

Следната функция прави алгоритъма бърз: намери ли съждителна променлива с две възможни стойности, разглежда само едната възможност (присвоява само стойност *истина* на променливата).

```
1. fillFirstAvailableAtomWithTrue():
2.      for i←1 to n
3.              if (disjuncts[getCode(makeDisjunct(i))]=FALSE) &&
                   (disjuncts[getCode(makeDisjunct(-i))]=FALSE) then
4.                      addDisjunct(makeDisjunct(i))
5.                      return TRUE
6.      return FALSE
```

Следващата функция е сърцевината на алгоритъма: тя реализира деветте стъпки от идейното описание.

```
1. sat3poly4(input): //returns 1 for solution, 0 for no solution and -1 for unknown
2.      for each D∈input:
3.              addDisjunct(D)
4.      hasArtificialAtoms←FALSE
5.      do
6.              while q.empty()=FALSE
7.                      resolution(q.pop())
8.              if disjuncts[getCode(makeEmptyDisjunct())]=TRUE then
9.                      if hasArtificialAtoms=FALSE then
10.                             return 0
11.                     else
12.                             return -1
13.             hasArtificialAtoms←TRUE
14.     while fillFirstAvailableAtomWithTrue()
15.     if hasContradiction() then
16.             return -1
17.     return 1 //the solution can be extracted from the global variable "disjuncts"
```

Помощната функция hasContradiction се извиква само когато всички съждителни променливи са получили конкретни стойности. Тя замества стойностите на променливите в дадената конюнктивна нормална форма, изчислява получения логически израз и връща отрицанието на изчислената стойност. Названието на функцията hasContradiction идва от очакването намерените стойности на променливите да удовлетворяват конюнктивната нормална форма; обратният резултат противоречи на това очакване. Функцията hasContradiction реализира стъпка 9 от идейното описание на алгоритъма.

Шестият и седмият ред пресмятат насищането.

Осмият ред от кода проверява дали е изведен празният дизюнкт — стъпки 2 и 6 от идейното описание. Флагът hasArtificialAtoms показва коя стъпка се изпълнява: стъпка 6 — когато hasArtificialAtoms е *истина*; стъпка 2 — когато hasArtificialAtoms е *лъжа*.

Следователно псевдокодът правилно реализира идейното описание на алгоритъма, чиято коректност беше обоснована по-горе.

Анализ на сложността на алгоритъма

Интересуваме се от времевата сложност на алгоритъма при най-лоши входни данни.

Всички дизюнкти са общо $1 + 2n + (2n)^2 + (2n)^3 = \Theta(n^3)$ и всеки от тях бива добавен в глобалната опашка q най-много веднъж. Всеки добавен в опашката дизюнкт се изважда от нея по някое време. При всяко изваждане на дизюнкт от опашката се извиква функцията resolution, а тя извиква не повече от три помощни функции (всъщност точно три за всеки подаден към нея непразен дизюнкт; празният се подава най-много веднъж). Затова трябва първо да установим колко време изразходват помощните функции, реализиращи деветте правила на 3-резолюцията.

Анализ на функциите, реализиращи деветте правила на 3-резолюцията:

Име на функцията	**Брой извиквания на функцията**	**Време, изразходвано при едно извикване**	**Общо изразходвано време**
resolution11	$O(n)$	$\Theta(1)$	$O(n)$
resolution12	$O(n)$	$\Theta(n)$	$O(n^2)$
resolution13	$O(n)$	$\Theta(n^2)$	$O(n^3)$
resolution21	$O(n^2)$	$\Theta(1)$	$O(n^2)$
resolution22	$O(n^2)$	$\Theta(n)$	$O(n^3)$
resolution23	$O(n^2)$	$\Theta(n^2)$	$O(n^4)$
resolution31	$O(n^3)$	$\Theta(1)$	$O(n^3)$
resolution32	$O(n^3)$	$\Theta(n)$	$O(n^4)$
resolution33	$O(n^3)$	$\Theta(n)$	$O(n^4)$

Събирайки времената на деветте функции, получаваме общо време $O(n^4)$ за пресмятане на разни 3-резолвенти, тоест за насищането на множеството от дизюнкти, изчислявано на редове 6 и 7 от кода на функцията sat3poly4.

За останалите редове от кода на функцията sat3poly4 получаваме следните оценки на времето:

Номер на ред от псевдокода	**Брой изпълнения на реда**	**Време, изразходвано при едно изпълнение**	**Общо изразходвано време**
редове 2 и 3	$O(n^3)$: толкова са 3-дизюнктите	$\Theta(1)$	$O(n^3)$
ред 4	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
редове 9–12	$\Theta(1)$: има само един празен дизюнкт	$\Theta(1)$	$\Theta(1)$
редове 8 и 13	$O(n)$: най-много веднъж за всяка съждителна променлива	$\Theta(1)$	$O(n)$
ред 14	$O(n)$: най-много веднъж за всяка съждителна променлива	$O(n)$	$O(n^2)$
ред 15	$O(1)$	$O(n^3)$: толкова са 3-дизюнктите	$O(n^3)$
редове 16 и 17	$O(1)$	$\Theta(1)$	$O(1)$

Следователно най-много време се изразходва за пресмятането на 3-резолвентите, тоест за насищането на даденото множество от 3-дизюнкти: $O(n^4)$. Това е и времевата сложност на целия алгоритъм sat3poly4.

Сложността по памет се определя от дължината на масива disjuncts и опашката q, а те зависят от броя на всевъзможните 3-дизюнкти на n съждителни променливи. Както установихме по-горе, въпросният брой е равен на $1 + 2n + (2n)^2 + (2n)^3 = \Theta(n^3)$. Получената количествена оценка е асимптотично точна за паметта, изразходвана от логическия масив disjuncts, защото той е статичен, тоест размерът му не зависи от броя на генерираните 3-дизюнкти. За опашката q това е само горна граница на използваното количество памет, защото в опашката се пазят само генерираните 3-дизюнкти. В крайна сметка сложността по памет на sat3poly4 се определя от масива disjuncts: тя е $\Theta(n^3)$ при всякакви входни данни.

Надеждност на полиномиалния алгоритъм sat3poly4

Някои конюнктивни нормални форми (от двата вида — както удовлетворими, така и неудовлетворими) алгоритъмът sat3poly4 не разпознава и връща отговор “неизвестно”. Два примера са показани в таблицата.

Конюнктивни нормални форми, с които полиномиалният алгоритъм не може да се справи	
Удовлетворима КНФ	Неудовлетворима КНФ
22 -21 1 22 -22 2 5 3 4 -5 1 2 6 3 -4 -6 1 2 7 -3 4 -7 1 2 8 -3 -4 -8 1 -2 9 3 4 -9 1 -2 10 3 -4 -10 1 -2 11 -3 4 -11 1 -2 12 -3 -4 -12 -1 2 13 3 4 -13 -1 2 14 3 -4 -14 -1 2 15 -3 4 -15 -1 2 16 -3 -4 -16 -1 -2 17 3 4 -17 -1 -2 18 3 -4 -18 -1 -2 19 -3 4 -19 -1 -2 20 -3 -4 -20 0 0 0	20 1 2 5 3 4 -5 1 2 6 3 -4 -6 1 2 7 -3 4 -7 1 2 8 -3 -4 -8 1 -2 9 3 4 -9 1 -2 10 3 -4 -10 1 -2 11 -3 4 -11 1 -2 12 -3 -4 -12 -1 2 13 3 4 -13 -1 2 14 3 -4 -14 -1 2 15 -3 4 -15 -1 2 16 -3 -4 -16 -1 -2 17 3 4 -17 -1 -2 18 3 -4 -18 -1 -2 19 -3 4 -19 -1 -2 20 -3 -4 -20 0 0 0

Първото число (самò на ред) е броят на съждителните променливи. Всеки следващ ред е един 3-дизюнкт: променливите са указани с поредните си номера (цели числа от 1 до максимума — числото от първия ред). Поначало номерата са цели положителни числа; знак минус пред номера означава, че съответната променлива участва с отрицание. Ред от три нули е признак за край на входните данни.

От тези примери става ясно, че има конюнктивни нормални форми, които полиномиалният алгоритъм не може да разпознае дали са удовлетворими, или неудовлетворими. Алгоритъмът би бил безполезен, ако твърде често връща отговор “неизвестно”. Така възниква въпросът колко често се получава този отговор.

Възможни са два подхода — теоретичен и експериментален. Теоретичният подход означава да се реши една комбинаторна или вероятностна задача (в зависимост от избрания математически инструментариум): колко са конюнктивните нормални форми (с известни характеристики — брой променливи и брой дизюнкти), които алгоритъмът не разпознава (или каква е вероятността случайно избрана КНФ да остане неразпозната)?

Експерименталният подход се състои в генериране на случайни КНФ и преброяване на неразпознатите. При опитите останаха неразпознати две от общо 72693 КНФ, тоест 0,003 %, с n променливи и d дизюнкта, където $20 \le n \le 30$ и $3n \le d \le 6n$.

Сравнение на sat3poly4 с други алгоритми

3-SAT	4-SAT	5-SAT	6-SAT	Тип	Публикация
1,782	1,835	1, 867	1, 888	дет.	[PPZ97]
1,618	1,839	1, 928	1, 966	дет.	[MS85]
1,588	1,682	1, 742	1, 782	ранд.	[PPZ97]
1,579	—	—	—	дет.	[Sch92]
1,505	—	—	—	дет.	[Kul99]
1,481	1,6	1, 667	1, 75	дет.	[DGH+02]
1,362	1,476	1, 569	1, 637	ранд.	[PPSZ98]
1,334	1,5	1, 6	1, 667	ранд.	[Sch99]
1,3302	—	—	—	ранд.	[HSSW02]
1,3290	—	—	—	ранд.	[BS03]
1,3280	—	—	—	ранд.	[Rol03]
1,324	1,474	—	—	ранд.	[*]

Таблицата съдържа времената на бързи експоненциални алгоритми (детерминирани и рандомизирани) за задачата 3-SAT и нейни варианти. Написани са само основите a на функциите a^n. Най-малкото $a = 1{,}324$.

Стандартен съвременен компютър има бързодействие $5 \cdot 10^{10}$ флопа. Ако ограничим времето до 1 сек., най-бързият експоненциален алгоритъм за 3-SAT ще се справи с $n \approx \log_{1{,}324}(5 \cdot 10^{10}) \approx 88$ променливи, а полиномиалният алгоритъм ще може да работи с $n \approx (5 \cdot 10^{10})^{1/4} \approx 473$ променливи, което е много повече, но не е достатъчно за големи практически задачи. (При сметките са пренебрегнати константните множители.)

Има и суперкомпютри с мощност около 10^{17} флопа [1]. На тях експоненциалният алгоритъм ще решава за 1 сек. КНФ с $n \approx \log_{1{,}324}(10^{17}) \approx 139$ променливи, а полиномиалният — с $n \approx (10^{17})^{1/4} \approx 17783$ променливи.

Алгоритъмът sat3poly4 използва памет $\Theta(n^3)$, затова на стандартен компютър с 16 гигабайта алгоритъмът ще решава КНФ с 2580 променливи, а на суперкомпютър с 512 терабайта [2] — дори КНФ с 82570 променливи. Следователно времето е по-силно ограничение от паметта.

Благодарности

Благодаря на доцент Стефан Герджиков от ФМИ на СУ "Свети Климент Охридски" за съветите му по реализацията на алгоритъма, спомогнали за намаляване на степенния показател на времевата сложност.

Литература

[1] https://en.wikipedia.org/wiki/Supercomputer

[2] https://www.nas.nasa.gov/hecc/resources/pleiades.html

[*] https://www.math.ucsd.edu/~sbuss/CourseWeb/Math268_2007WS/IwamaTamaki03.pdf?fbclid=IwAR03SSYwnF6dDdxHMNi9ru0r0w75EKK1fLwCdHWE2rwwLUbd77LosJTh138

[BS03] S. Baumer and R. Schuler. Improving a probabilistic 3-SAT Algorithm by Dynamic Search and Independent Clause Pairs. ECCC TR03-010, 2003. Also presented at SAT 2003.

[DGH+02] E. Dantsin, A. Goerdt, E. A. Hirsch, R. Kannan, J. Kleinberg, C. Papadimitriou, P. Raghavan, and U. Schöning. A deterministic $2 - \frac{2}{k+1}$ algorithm for k-SAT based on local search. TCS, 289(1):69–83, 2002.

[HSSW02] T. Hofmeister, U. Schöning, R. Schuler and O. Watanabe. Probabilistic 3-SAT Algorithm Further Improved. Proc. 19th STACS, LNCS 2285, 2002.

[Kul99] O. Kullmann, New methods for 3-SAT decision and worst-case analysis. TCS, 223(1-2):1–72, 1999.

[MS85] B. Monien and E. Speckenmeyer. Solving satisfiability less than 2^n steps. Discrete Appl. Math., 10:287–295, 1985.

[PPSZ98] R. Paturi, P. Pudlák, M. E. Saks, and F. Zane. An improved exponential-time algorithm for k-SAT. Proc. 39th FOCS, 1998.

[PPZ97] R. Paturi, P. Pudlák, and F. Zane. Satisfiability coding lemma. Proc. 38th FOCS, 1997.

[Rol03] D. Rolf. 3-SAT $\in$ RTIME(O(1.32793n)). ECCC TR03-054, 2003.

[Sch92] I. Schiermeyer. Solving 3-Satisfiability in less than 1.579^n steps. CSL 92, LNCS 702, 1992.

[Sch99] U. Schöning. A probabilistic algorithm for k-SAT and constraint satisfaction problems. Proc. 40th FOCS, 1999.