МАТЕМАТИКА И МАТЕМАТИЧЕСКО ОБРАЗОВАНИЕ, 2021
MATHEMATICS AND EDUCATION IN MATHEMATICS, 2021
Proceedings of the Fiftieth Spring Conference
of the Union of Bulgarian Mathematicians
2021

50 ГОДИНИ ОТ ПУБЛИКУВАНЕТО НА ТЕОРЕМАТА НА КУК*

Красимир Манев

Алгоритмите са същността на Информатиката, както и на някои класически дялове на математиката. Но до 1930-те години алгоритмите се разглеждат само като инструмент за решаване на важни за науката и практиката задачи. Тъй като понятието **не е формално**, едва откъм средата на 1930-те години на тях започва да се гледа като на обект за изследване и да се поставят редица съществени въпроси. Например, може ли понятието *алгоритъм* да се формализира? Какви задачи могат да се решават алгоритмично и има ли задачи, които не могат да се решат с алгоритъм? Защо различните алгоритми за една и съща задача работят за съществено различно време? Затова можем смело да кажем, че през 30-те години на миналия век се поставят основите на Математическата теория на алгоритмите, която е предмет на настоящата статия. През 2021 г. се навършват 50 години от публикуването на много важен резултат на тази теория. Изложението в статията е съсредоточено върху този резултат и последвалото от него предизвикателство, което не е получило още отговор.

1. Въведение. Човечеството използва **алгоритми** от дълбока древност. Не е случайно, че повечето тълковни речници дават като най-популярен пример за този род процедури *Алгоритъма на Евклид*, публикуван в книгата му „Елементи“, около 300 години преди новата ера. Може да се предположи, че алгоритмични процедури са били използвани и преди това. Редица такива процедури, разработени от древните азиатски учени са достигнали до Европа благодарение на трите трактата на арабския математик Мухаммад ибн Муса ал Хорезми (от името на когото е извлечена думата алгоритъм), двата най-популярни от които са *За смятането с индийски числа* (съдържащ алгоритмите за аритметични операции с естествени числа, записани в позиционна бройна система, които използваме и до днес) и *За смятането с приведение (ал-джабър, б.а.) и двустранно отстраняване (ал-мукабала, б.а.)* (съдържащ алгоритми за решаване на линейни уравнения, от името на който произлиза названието Алгебра).

Огромният интерес към алгоритмите безусловно се дължи на широкото разпространение в наши дни на универсалните компютри. Но предчувствието за появата на машина с програмно управление, способна да изпълнява зададен ѝ алгоритъм, витае във въздуха в периода между двете световни войни. По това време много математици започват да търсят отговори на важните въпроси, споменати по-горе. В резултат на тези изследвания са създадени значителен брой *формализми*, моделиращи

*__ACM Classification Codes:__ F.1.1, I.1.2, F.2.

неформалното понятие *алгоритъм* – Машина на Пост [6], Машина на Тюринг [7], λ-смятане [2], Рекурсивни функции [5], Нормални алгоритми на Марков [9], Формални граматики от общ тип [1]. С доказване на еквивалентността на всички тези формализми беше решен въпросът *Каква е математическата същност на неформалното понятие алгоритъм?*, а в светлината на работите на К. Гьодел беше показано и съществуването на достатъчно просто формулирани задачи, които не могат да бъдат решени алгоритмично.

След като имаме избран някакъв формализъм, с който моделираме математически неформалното понятие *алгоритъм*, можем да изберем и формален математически обект, с който да моделираме неформалното понятие *задача*, без което математическото изследване на алгоритмите е невъзможно. Естественото формално понятие в случая е *функция*, чиито стойности могат да бъдат изчислявани/намирани със средствата на избрания формализъм – такива функции в теорията се наричат *изчислими*.

Следващ важен етап в развитието на областта е дефинирането на понятията (зависещи от формализма) *сложност на алгоритъм A по време или памет*, в най-лошия (и в средния) случай. Това са функции на променлива $n \in N$, където с N означаваме естествените числа, оценяващи броя направени стъпки на съответния формализъм, или броя единици използвана памет, при работа на A върху най-трудния за обработване вход на алгоритъма с размер n (или усреднено по всички входове с размер n). Така беше даден отговор и на въпроса *Защо различните алгоритми за една и съща задача работят с различна скорост и се нуждаят от различен обем памет?*. За програмистката практика отговорът на тези въпроси беше безценен, тъй като разработчиците на програми започнаха да обръщат много повече внимание на бързодействието на използваните алгоритми и икономията на памет.

Наличието на няколко алгоритъма за решаване на една и съща задача постави следващия важен въпрос на теорията – *Кой е алгоритъмът с най-добра сложност от няколко възможни?* За съжаление, доказателства за долната граница на сложността по време имаме за алгоритмите на много малко от интересните задачи. Затова усилията на действащите изследователи в областта, в последните години на XX век и досега, са насочени към изясняване на понятието *сложност на алгоритмично разрешимите задачи*. На този важен клон на Теорията на алгоритмите и на навършващите се 50 години от публикуването на един фундаментален резултат в това направление е посветена настоящата статия.

2. Основни понятия. Нека $C = \{a_1, a_2, \ldots, a_n\}$ е крайно множество, което наричаме *азбука*, а елементите му — *букви*. Наредената последователност $\alpha = a_{i_1}$, $a_{i_2}, \ldots a_{i_l}$ от букви на азбуката C наричаме *дума* над C, а цялото неотрицателно $l = |\alpha|$ — *дължина* на α. С ε означаваме думата, която не съдържа букви — *празната дума*, $|\varepsilon| = 0$. Празната дума е дума над всяка азбука.

Нека с C^l означим множеството от думите над C с дължина $l, l = 0, 1, 2, \ldots$. С C^* означаваме множеството от всички думи над азбуката C: $C^* = C^0 \cup C^1 \cup C^2 \cup \ldots$. Всяко множество от думи $L \subseteq C^*$ наричаме *език* над C. Задачата: „За дадени език $L \subseteq C^*$ и дума $\alpha \in C^*$ да се провери дали $\alpha \in L$.“ наричаме *задача за разпознаване на език*. Например, първата задача на всеки компилатор е по зададена програма (дума над клавиатурната азбука) да провери дали тя е от езика му (за програмиране)!

Нека $B_2 = \{0,1\}$, а B_2^n е множеството от всички n-мерни двоични вектори $(b_1, b_2, \ldots, b_n)$, където $b_i \in B_2$, $i = 1, 2, \ldots, n$. Функцията $\phi : B_2^n \to B_2$ наричаме *двоична* (или *булева*) функция на вектора от n променливи $\widetilde{x} = (x_1, x_2, \ldots, x_n)$, за всяко $n = 1, 2, 3, \ldots$. Стойностите на функцията означаваме с $\phi(\widetilde{x}) = \phi(x_1, x_2, \ldots, x_n)$.

Теорема 2.1. $|\{\phi | \phi : B_2^n \to B_2\}| = 2^{2^n}$.

Съгласно Теорема 2.1 булевите функции на една променлива са 4: двете *константи* $\phi_{1,0}(x) \equiv 0$ и $\phi_{1,3}(x) \equiv 1$, *идентитетът* $\phi_{1,1}(x) = x$ и *отрицанието* $\phi_{1,2}(x) = \overline{x}$, където $\overline{x} = 1$, ако $x = 0$ и $\overline{x} = 0$, ако $x = 1$. Двете константи означаваме винаги с $\widetilde{0}$ и $\widetilde{1}$, независимо като функции на колко променливи ги разглеждаме. От 16-те булеви функции на две променливи, за изложението ще ни трябват, освен двете константи, функциите *конюнкция* $\phi_{2,1}(x, y) = x \wedge y$, за която $x \wedge y = 1$ тогава и само тогава, когато[1] $x = y = 1$, и *дизюнкция* $\phi_{2,7}(x, y) = x \vee y$, , за която $x \vee y = 0$ т.с.т.к. $x = y = 0$. С x^σ означаваме x, когато $\sigma = 0$ или $\overline{x}$, когато $\sigma = 1$.

По аналогия с аритметичните изрази, в алгебрата на булевите функции можем да строим булеви *изрази/формули*, в които участват променливи, константи и знаците за отрицание (като 1-местна операция), конюнкция и дизюнкция (като 2-местни операции). Приоритетът на отрицанието в булевите формули е най-висок, следван от приоритета на конюнкцията, а дизюнкцията е с най-нисък приоритет.

Теорема 2.2 (Дж. Бул)**.** *Всяка булева функция може да се представи с формула над отрицанието, конюнкцията и дизюнкцията.*

Без да се впускаме в доказателство, просто ще покажем едно възможно представяне на всяка булева функция с операциите от Теорема 2.2. Ако $\phi(\widetilde{x}) = \widetilde{1}$, тогава $\widetilde{1} = x_1 \vee \overline{x_1}$. Всяка друга функция можем да представим с каноничната формула, наричана *съвършена конюнктивна нормална форма* (СКНФ):

$$\phi(x_1, x_2, \ldots, x_n) = \bigwedge_{\substack{\forall(\sigma_1\sigma_2\ldots\sigma_n) \\ \phi(\sigma_1\sigma_2\ldots\sigma_n)=0}} (x_1^{\sigma_1} \vee x_2^{\sigma_2} \vee \ldots \vee x_n^{\sigma_n}).$$

Доказателството следва от факта, че *елементарната дизюнкция* $x_1^{\sigma_1} \vee x_2^{\sigma_2} \vee \ldots \vee x_n^{\sigma_n}$ ще има стойност 0, т.с.т.к $x_1 = \sigma_1, x_2 = \sigma_2, \ldots, x_n = \sigma_n$. За пример да разгледаме булевата функция на три поменливи $\phi(x_1, x_2, x_3)$, която има нулеви стойности за векторите $(0,1,0), (0,1,1)$ и $(1,0,1)$. Тъй като функцията не е константата $\widetilde{1}$, тя се представя със СКНФ:

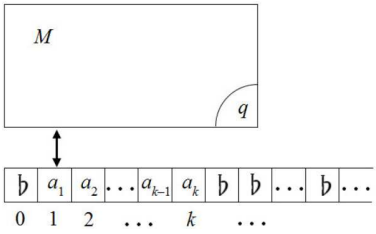
$$\begin{aligned} \phi(x_1, x_2, x_3) &= (x_1^0 \vee x_2^1 \vee x_3^0) \wedge (x_1^0 \vee x_2^1 \vee x_3^1) \wedge (x_1^1 \vee x_2^0 \vee x_3^1) = \\ &= (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3}) \end{aligned}$$

За някои функции, СКНФ може да се упрости до *конюнктивна нормална форма* (КНФ), в която не всяка елементарна дизюнкция съдържа всички променливи. Някои БФ имат по няколко различни КНФ.

3. Машина на Тюринг (МТ). Както вече стана дума по-горе, за да можем да изследваме математически неформалните понятия алгоритъм и задача, трябва да ги моделираме със съответни математически обекти. За целите на това изложение като формализъм за понятието алгоритъм ще използваме най-удобния за целта – Машина на Тюринг (МТ), а за задача – изчислима с МТ функция и разпознаван от МТ език. Машината на Тюринг е абстрактна математическа машина $M = \langle Q, X, \delta, q_0, F \rangle$,

[1] по-нататък ще използваме съкращението (т.с.т.к.)

представена схематично на Фиг. 1.



Фиг. 1. Машина на Тюринг

Във всеки един момент от работата си M може да се намира в точно едно състояние от крайното множество от *състояния* Q, като винаги започва работа в началното състояние $q_0 \in Q$. Машината има памет – безкрайна в едната посока *лента*, разделена на *клетки*. Всяка клетка на лентата може да съхранява една от буквите на крайната *азбука* X. Азбуката непременно съдържа една специална буква $\flat$ – *бленк*, ролята на която е да запълва клетките на лентата, които не са използвани. Във всеки момент от работата на машината на лентата има само краен брой клетки, в които има различни от бленк букви. Думата $\alpha \in X^*$, започваща от клетка 1 на лентата и завършваща с последната различна от бленк буква наричаме *текуща лентова дума*.

Четящо-пишещата *глава* на машината във всеки момент от работата се намира точно върху една от клетките на лентата и може да прочете намиращата се там буква. Работата на машината се управлява от функцията $\delta : Q \times X \to Q \times X \times \{-1, 0, 1\}$, като $\delta(q, x) = (q', x', m)$ означава, че ако в момент t на дискретна целочислена скала на времето $T = \{0, 1, 2, \ldots\}$ машината се намира в състояние $q \in Q$ и чете от лентата буквата $x \in X$, тогава тя:

• ще запише на лентата $x' \in X$;

• ще премести главата една клетка в посока началото, ако $m = -1$, една клетка в обратната посока, ако $m = 1$, или ще остане над същата клетка, ако $m = 0$;

• в следващия момент на времето $t + 1$ ще бъде в състояние $q' \in Q$.

Функцията δ всъщност е това, което можем да наречем *програма* за МТ. А текущото състояние на МТ и текущата лентова дума наричаме *конфигурация*.

Състоянията от $F \subset Q$ наричаме *заключителни състояния*. Попадайки в едно от заключителните състояния, машината прекратява работата. Процесът, започващ в начално състояние и завършващ със спиране в някое от заключителните състояния, наричаме *изчисление*. Разбира се, както при всяко формално изчисление, и при МТ е възможно при някои начални лентови думи машината никога да не стигне до заключително състояние, т.е. **да зацикли**. Тук няма да се занимаваме с изчисления, които зациклят.

За целите ни са интересни два вида завършващи изчисления с МТ. Нека машината стартира с лентова дума $\alpha \in X_f^*$, където X_f^* е множеството от думи, с които *кодираме* стойностите в дефиниционната област на функцията $f : X_f^* \to X^*$, и завършва работа в заключително състояние, с лентова дума $\beta = f(\alpha)$. Тогава казваме, че машината *пресмята функцията* f. Такива функции наричаме *изчислими по Тюринг*.

Втората възможност е да разглеждаме машини, за които $F = \{q_Y, q_N\}$. Нека $L \subseteq X^*$ е език над X и машината спира в състояние q_Y, когато работи върху лентова дума $\alpha \in L$, а спира в състояние q_N, когато работи върху лентова дума $\alpha' \notin L$. В такъв случай казваме, че машината *разпознава езика* L и наричаме L език, *разпознаван* от МТ. Този вид изчисление не е принципно различен от изчисляването на функция, тъй като можем да си мислим за него като за пресмятане на функция

$f_L : X^* \to \{true, false\}$, за която $f_L(\alpha) = true$ т.с.т.к. $\alpha \in L$.

Тук трябва да подчертаем още веднъж, че МТ и всеки друг от споменатите погоре формални модели за изчисление изчисляват едни и същи функции и разпознават едни и същи езици. Това се отнася и за компютрите, които са реални **крайни** еквиваленти на споменатите формализми (и за универсалните езици за програмиране). Разликата е, че програмите за МТ се пишат трудно и една практична програма за МТ би била с огромен обем. В същото време, компютърният език и езиците за програмиране са много сложни обекти, трудно приложими за доказване на математически свойства, което при МТ, както ще видим по-нататък, е много по-лесно.

4. Сложност на алгоритъм. Представянето на резултата, който е обект на тази публикация, не е възможно без познаването на понятията *сложност* на алгоритъм *по време* и *по памет*. Затова накратко да представим това важно понятие в неговия **машинно-зависим вариант**.

Нека A е програма за МТ, решаваща задачата f с домен X_f^*, без значение дали f е за изчисляване на функция или за разпознаване на език. Да означим с $T_A(\alpha)$ броя стъпки, които A прави при работа върху входната дума α, а с $S_A(\alpha)$ – броя клетки, които A използва при работа върху α. Сложност по време и памет на алгоритъма A, **в най-лошия случай**, наричаме следните две функции на променливата n, която наричаме *размер на входа*:

$$t_A(n) = \max_{\forall \alpha \in X_f^*, |\alpha|=n} T_A(\alpha), \quad s_A(n) = \max_{\forall \alpha \in X_f^*, |\alpha|=n} S_A(\alpha).$$

Между *сложността по време* и *сложността по памет* в най-лошия случай съществува следната важна връзка:

Лема 4.1. *За всяка програма A и за всяко n, $s_A(n) \leq t_A(n)$.*

Твърдението е очевидно, защото не е възможно, която и да е входната дума α, за $T_A(\alpha)$ стъпки да бъдат използвани повече от $S_A(\alpha)$ клетки на лентата.

Тъй като размерът на входа е от съществена важност за получаваната функция на сложност, на това място в теорията непременно трябва да се постави условието за **разумно кодиране** на входа. Неформално, това ще рече да се кодират входните данни на алгоритъма пестеливо с думи от X^*, с колкото може по-малко букви, без да се губи разбира се наличната във входните данни информация. Така например, ако входът на задачата е цяло число, не е редно да го задаваме на алгоритъма предшествано от произволен брой незначещи нули.

Както се вижда, функциите на сложност на алгоритмите са дискретни/целочислени и в този смисъл са неудобни за пряко използване в математически разсъждения. Например, за сравняване на растежа им. Затова, вместо със самите функции на сложност си служим с **асимптотични оценки** на тяхното поведение. Да означим с $F_{N->R}$ множеството на функциите с аргументи естествени числа и реални стойности.

Нека $u \in F_{N->R}$ и $v \in F_{N->R}$. Казваме, че $u \in O(v)$, ако съществуват $n_0 \in N$ и $c \in R, c > 0$ такива, че $u(n) \leq cv(n)$, $\forall n \in N, n \geq n_0$. Лесно се вижда, че $u \in O(u)$, и че от $u \in O(v), v \in O(w)$ следва $u \in O(w)$, т.е. релацията $\in O()$ е рефлексивна и транзитивна, но не е симетрична, нито антисиметрична. Например, $n \in O(n^2)$, а $n^2 \notin O(n)$, докато $n^2 \in O(n^2 + 1)$ и $n^2 + 1 \in O(n^2)$.

В същото време $O(u) = O(v)$ т.с.т.к. $u \in O(v)$ и $v \in O(u)$ и можем да дефинираме релацията $r_{\asymp} \subseteq F_{N->R} \times F_{N->R}$ с $r_{\asymp} = \{(u, v) | u \in O(v), v \in O(u)\}$. От доказаното

по-горе следва, че $r_{\asymp}$ е релация на еквивалентност и разбива $F_{N->R}$ на класове на еквивалентност. Класът на u означаваме с $\Theta(u)$ и наричаме *порядък на растеж* на u. Очевидно $\Theta(u) \subseteq O(u)$.

Нека $\mathcal{O}$ се състои от всички различни класове на еквивалентност на релацията $r_{\asymp}$. Релацията $r_{\preceq} \subseteq \mathcal{O} \times \mathcal{O}$ дефинираме с $r_{\preceq} = \{(\Theta(f), \Theta(g)) |\ f \in O(g)\}$. Ако $X \preceq Y$ и $X \neq Y$, тогава пишем $X \prec Y$. Тази релация е частична наредба и ни позволява да сравняваме порядъците на растеж на сложностите на различни алгоритми. Например, популярните алгоритми за сортиране са от два класа – $\Theta(n^2)$ и $\Theta(n \log n)$. Тъй като $\Theta(n \log n) \prec \Theta(n^2)$, алгоритмите от първия вид са за предпочитане.

За оценка на обичайните алгоритми, изучавани в различни дисциплини на информатиката, се използват асимптотичните класове: $O(1)$ (константна сложност) $\prec O(\log n)$ (логаритмична сложност) $\prec O(n)$ (линейна сложност)$\prec O(n \log n) \prec O(n^2)$ (квадратна сложност) $\prec O(2^n)$ (експоненциална сложност) и т.н. В общия случай, алгоритмите с асимптотична сложност $O(n^d)$ (както и такива със сложност $O(n^d \log^c n)$, доколкото $O(\log n) \prec O(n)$) наричаме *полиномиални*. В програмистката практика определението „полиномиален“ е синоним на „добър“ (в практиката много рядко се срещат полиномиални алгоритми с голямо d).

Макар и не съвсем издържано от математическа гледна точка, почти навсякъде в литературата, вместо използваната по-горе нотация, по традиция пишем $t(n) = O(g(n))$ или $t(n) = \Theta(g(n))$.

5. Сложност на задачи и Теорема на Кук. След като имаме добре дефинирано понятие за сложност на алгоритъм, би било интересно да определим по някакъв начин и *сложност на задача* например като асимптотичната оценка на **най-добрия алгоритъм**, който решава задачата. За съжаление, както вече споменахме по-горе, доказани точни долни граници на сложността на алгоритмите имаме за много малко задачи. Този проблем занимава теоретиците в областта в края на 60-те години на миналия век. Значителен принос в това направление са резултатите на американския математик Стивън Артър Кук[2].

В програмистката практика са известни много задачи, за които има един или повече алгоритми с полиномиална сложност. Със сигурност има и задачи, за които съществуват алгоритми с полиномиална сложност, без да са ни все още известни. Пример за такава задача е Задачата на линейното програмиране, за която дълго не се знаеше има ли полиномиален алгоритъм, а се оказа, че такъв съществува. Да означим с $\mathcal{P}$ класа от **всички задачи**, за които съществуват алгоритми с полиномиална сложност, дори без да са ни известни.

За изследване обсега на $\mathcal{P}$ и неговата „околност“ от задачи се оказва полезно да се ограничим само със задачите за разпознаване на език. Всъщност всяка задача за пресмятане на изчислима функция „Пресметнете стойността на $f(\alpha)$!“ можем да трансформираме в подобна задача за разпознаване на език „Равна ли е стойността $f(\alpha)$ на B?“, където B е елемент от областта на изменение на функцията f. А задача за намиране на оптимум „Намерете максимум/минимум на f!“ можем да трансформираме в задача за разпознаване на език „Има ли стойност на f по-голяма/по-малка

[2]Ст. Кук (Stephen Arthur Cook) е роден през 1939 г. През 1966 той защитава докторат в Харвард и започва работа в Университета на Калифорния в Бъркли. Парадоксално, месеци преди да излезе публикацията [3], която е тема на тази статия и която го прави световно известен, той е освободен от университета, като безперспективен.

от B?“. Интуитивното основание за това редуциране е, че задачата за разпознаване на език, която получаваме по този начин, в никакъв случай не е с по-голяма сложност от съответната задача за пресмятане на функция.

Нека f и g са задачи за разпознаване на език. Казваме, че f се *свежда полиномиално* към g, ако съществува полиномиален алгоритъм A, който трансформира всяка стойност α от дефиниционната област на f до стойност α' от дефиниционната област на g така, че $f(\alpha) = true \iff g(A(\alpha)) = true$. Ще означаваме с $f \propto g$ полиномиалната сводимост на f към g. Важността на релацията $\propto$ се вижда от следните свойства:

Лема 5.1. *Ако $f \propto g$, от $g \in \mathcal{P}$ следва $f \in \mathcal{P}$ (или еквивалентното от $f \notin \mathcal{P}$ следва $g \notin \mathcal{P}$).*

Лема 5.2. *Ако $f \propto g$ и $g \propto h$, то $f \propto h$.*

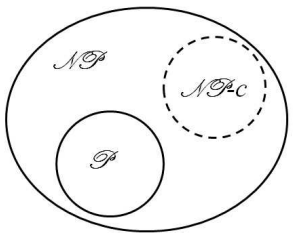
Задачата за разпознаване на език f наричаме *недетерминирано-полиномиална*, ако МТ може да разреши полиномиално всеки неин екземпляр с помощта на хипотетичен обект, наричан *оракул* или *демон*, който по време на работата „помага“ на МТ. От практическа гледна точка можем да приемем, че оракулът подсказва на МТ някаква информация, която наричаме *догадка*. Затова е по-удобно да наречем една задача за разпознаване на език *полиномиално-проверима*, ако съществува полиномиален алгоритъм, разпознаващ езика f с помощта на подсказана от оракула догадка β_α, за всяка дума α от домена на f. Означаваме с $\mathcal{NP}$ класа на полиномиално-проверимите задачи (т.е. недетерминирано-полиномиалните) задачи.

Полиномиалната проверимост на една задача се установява сравнително лесно. Да разгледаме задачата Хамилтонов цикъл: „Даден е граф. Има ли в него Хамилтонов цикъл?“. Не е известен полиномиален алгоритъм за решаване на тази задача. Проверката, обаче, дали зададена редица от върхове на графа (догадка) образува Хамилтонов цикъл е елементарна. Достатъчно е да преминем редицата от началото към края и да проверим дали всеки два съседни върха образуват ребро, дали началото и краят ѝ съвпадат и дали всеки връх се среща точно веднъж. Тази проверка ще отнеме време $O(n)$.

Очевидно $\mathcal{P} \subseteq \mathcal{NP}$. За полиномиалната проверимост на всяка задача от $\mathcal{P}$ е достатъчно да я решим за полиномиално време (като не се налага да използваме думата *догадка*). Генералният въпрос е дали $\mathcal{P}$ е **същинско подмножество** на $\mathcal{NP}$ или не. Засега вярността на всяка една от двете хипотези е възможна, макар че от практически съображения по-вероятна изглежда първата възможност (това е и предположението на Кук). Затова останалите разглеждания в този раздел ще направим при предположението, че $\mathcal{P} \neq \mathcal{NP}$.

Лема 5.3. *Нека $f, g \in \mathcal{NP}$, $f \in \mathcal{NP}$-с и $f \propto g$. Тогава $g \in \mathcal{NP}$-с.*

Задача f, за която: а) $f \in \mathcal{NP}$; б) $\forall g \in \mathcal{NP}, g \propto f$, наричаме *$\mathcal{NP}$-пълна*. Означаваме с $\mathcal{NP}$-с класа на $\mathcal{NP}$-пълните задачи. При предположението, че $\mathcal{P} \neq \mathcal{NP}$, взаимното разположение на $\mathcal{P}$, $\mathcal{NP}$ и $\mathcal{NP}$-с е показано на Фиг. 2.



Фиг. 2. Класове на сложност

Ако имаме поне една $\mathcal{NP}$-пълна задача f, тогава с полиномиално свеждане на тази задача към подходяща задача g можем да покажем, че и g е $\mathcal{NP}$-пълна. Условието от дефиницията: вся-

ка задача от $\mathcal{NP}$ да се свежда полиномиално към $f \in \mathcal{NP}$, прави намирането на поне една $\mathcal{NP}$-пълна задача f много трудно. И все пак! Да разгледаме задачата:

Удовлетворимост на БФ: Дадена е булева функция $\phi(x_1, x_2, \ldots, x_n)$ в КНФ. Съществува ли вектор от стойности на променливите $(\sigma_1, \sigma_2, \ldots, \sigma_n)$ такъв, че $\phi(\sigma_1, \sigma_2, \ldots, \sigma_n) = 1$? Ще казваме, че такъв вектор $(\sigma_1, \sigma_2, \ldots, \sigma_n)$ *удовлетворява* ϕ.

Теорема 5.4 (Кук-Левин[3]). *Задачата* Удовлетворимост на БФ *е $\mathcal{NP}$-пълна* [3].

Доказателство.[4] 1) Лесно се проверява, че Удовлетворимост на БФ е от $\mathcal{NP}$. За да проверим, че даден вектор-догадка β удовлетворява функцията ϕ, е достатъчно да заместим всяка променлива със съответната стойност от вектора. Ако всяка елементарна дизюнкция приема стойност 1 върху α, тогава той удовлетворява ϕ. Тази проверка изисква време пропорционално на дължината на КНФ, т.е. сложността ѝ е полином (от първа степен) на дължината на входа.

2) Остава да покажем, че всяка друга задача от $\mathcal{NP}$ се свежда полиномиално към Удовлетворимост на БФ. Нека f е произволна задача от $\mathcal{NP}$, т.е. съществува машина на Тюринг $M =< Q, X, q_0, \delta, F >$, която за полиномиално време проверява f. Нека $Q = \{q_0, q_1 = q_Y, q_2 = q_N, \ldots, q_r\}$, $X = \{x_0 = \flat, x_1, \ldots, x_s\}$. Да означим с L_M езика от всички входни думи, за които M дава положителен отговор. Нека $\alpha = x_{i_1}, x_{i_2}, \ldots, x_{i_n}$ е дума от азбуката на M. Нека $t_M(n) \in \Theta(p(n))$, където $p(n)$ е полином. Без ограничение на общността ще считаме, че $t_M(n) = p(n)$. Ще посочим полиномиален алгоритъм, който от M и входна дума α на задачата f образува БФ ϕ в КНФ такава, че $\alpha \in L_M$ т.с.т.к. съществува вектор σ от стойности на променливите на ϕ и $\phi(\sigma) = 1$.

Ще използваме следните групи от променливи за построяване на булевата функция:

– $Q[i,k], 1 \leq i \leq p(n), 0 \leq k \leq r$ – приема стойност единица т.с.т.к в момент i машината е била в състояние q_k.

– $H[i,j], 1 \leq i \leq p(n), 1 \leq j \leq p(n)$ – приема стойност единица т.с.т.к в момент i главата на машината е била над клетка с номер j.

– $S[i,j,l], 1 \leq i \leq p(n), 1 \leq j \leq p(n), 0 \leq l \leq s$ – приема стойност единица т.с.т.к в момент i в клетката с номер j се е намирала буквата x_l.

Ще построим шест групи от елементарни дизюнкции, които съставят исканата КНФ така, че от конюнкцията на тези дизюнкции да се вижда ясно, че това е функцията, която търсим.

1) Във всеки дискретен момент M се намира в точно едно състояние.

Изискването M да бъде в поне едно състояние, за всяко i, $1 \leq i \leq p(n)$, отразяваме в елементарната дизюнкция

$$Q[i,0] \vee Q[i,1] \vee \ldots \vee Q[i,r],$$

а за да не бъде в повече от едно състояние, за всяко i, $1 \leq i \leq p(n)$, и за всяка двойка k, k', $0 \leq k < k' \leq r$, създаваме елементарна дизюнкция

$$\overline{Q[i,k]} \vee \overline{Q[i,k']} = \overline{Q[i,k] \wedge Q[i,k']}.$$

[3]Еквивалентно твърдение е доказано, независимо от Ст. Кук, от съветския математик Л. А. Левин [8], но тъй като резултатът на Левин е публикуван по-късно, теоремата е по-известна като Теорема на Кук.

[4]Доказателството следва изложението в [4].

Изразът вдясно на равенството не е елементарна дизюнкция, но от него по-лесно се вижда предназначенвнието му — не е вярно, че в кой да е момент МТ може да се намира в повече от едно състояние. Построяването на тези дизюнкции внася в конюктивната форма $p(n)(r+1) + 2p(n)\binom{r+1}{2} = p(n)(r+1)^2$ букви.

2) Във всеки момент главата на M се намира точно над една клетка.

В тази група подобно на първата строим за всяко i, $1 \leq i \leq p(n)$, дизюнкциите

$$H[i,1] \vee H[i,2] \vee \ldots \vee H[i,p(n)]$$

и

$$\overline{H[i,j]} \vee \overline{H[i,j']}, 1 \leq j < j' \leq p(n).$$

Построяването на тези дизюнкции внася в конюктивната форма $p(n)^2 + 2p(n)\binom{p(n)}{2}$ $= p(n)^3$ букви.

3) Във всеки момент, във всяка от клетките, които могат да бъдат достигнати от главата на M, се намира точно една буква.

Тази група строим както първите две, като за всяко i, $1 \leq i \leq p(n)$, и за всяко j, $1 \leq j \leq p(n)$, включваме дизюнкциите

$$S[i,j,0] \vee S[i,j,1] \vee \ldots \vee S[i,j,s]$$

и

$$\overline{S[i,j,l]} \vee \overline{S[i,j,l']}, 0 \leq l < l' \leq s.$$

Построяването на тези дизюнкции внася в конюктивната форма $p(n)^2(s+1) + 2p(n)^2\binom{s+1}{2} = p(n)^2(s+1)^2$ букви.

4) В началния момент машината M е готова да започне работа върху думата ω.

В началния момент M се намира в състояние q_0, главата е над клетката с номер 1, в клетките с номера от 1 до n е записана думата ω, а клетките с номера от $n+1$ до $p(n)$ са запълнени с бленкове. Тази ситуация представяме със следните $p(n)+2$ еднобуквени дизюнкции

$$Q[0,0], H[0,1], S[0,1,i_1], \ldots, S[0,n,i_n], S[0,n+1,0], \ldots, S[0,p(n),0].$$

5) След не повече от $p(n)$ стъпки машината ще бъде в допускащото заключително състояние $q_Y = q_1$ (може да приемем, че след достигане на заключително състояние, МТ ще спре да променя конфигурацията до момента $p(n)$).

Тази група съдържа само една еднобуквена дизюнкция – $Q[p(n),1]$.

6) За всеки момент i, $0 \leq i \leq p(n)$, конфигурацията на M в момента $i+1$ се получава от конфигурацията ѝ в момента i по правилата, определени от δ.

Тази група е най-сложна. Разделяме я на две подгрупи от елементарни дизюнкции. В първата група са дизюнкциите, които ни гарантират, че не са възможни изменения, нерегламентирани с функцията на преходите, т.е. не е възможно главата да не е над клетка j в момент i, а в момент $i+1$ съдържанието на тази клетка да се е променило. За всяко $i, 1 \leq i \leq p(n)$, за всяко $j, 1 \leq j \leq p(n)$, и за всяко $l, 0 \leq l \leq s$, построяваме дизюнкцията

$$\overline{S[i,j,l]} \vee H[i,j] \vee S[i+1,j,l] = \overline{S[i,j,l] \wedge \overline{H[i,j]} \wedge \overline{S[i+1,j,l]}}.$$

От тази група в конюнктивната форма се добавят $3p(n)^2(s+1)$ букви.

Втората група се грижи за това, измененията на конфигурациите да се извършват по правилата, зададени от δ. За всяка четворка (i,j,k,l), $1 \leq i \leq p(n)$, $1 \leq j \leq p(n)$, $0 \leq k \leq r$ $l, 0 \leq l \leq s$, в тази подгрупа поставяме следните три елементарни дизюнкции:

$$\overline{H[i,j]} \vee \overline{Q[i,k]} \vee \overline{S[i,j,l]} \vee H[i+1, j+m],$$
$$\overline{H[i,j]} \vee \overline{Q[i,k]} \vee \overline{S[i,j,l]} \vee Q[i+1, k'],$$
$$\overline{H[i,j]} \vee \overline{Q[i,k]} \vee \overline{S[i,j,l]} \vee S[i+1, j, l'],$$

където $\delta(q_k, x_l) = (q_{k'}, x_{l'}, m)$, $m = -1, 0, 1$, ако $q \in Q$. Ако $q \in \{q_y, q_n\}$, тогава $k' = k, l' = l$ и $m = 0$.

От тази подгрупа в КНФ ще влязат $12p(n)^2(r+1)(s+1)$ букви.

Да свържем с конюнкции всички построени дизюнкции. Получаваме конюнктивна нормална форма на булева функция ϕ. Така, на всеки екземпляр α на задачата, решавана от M за полиномиално време, съпоставяме екземпляр на задачата Удовлетворимост на БФ. При това лесно се вижда, че M завършва в q_Y при работа върху α т.с.т.к. съществува вектор от стойности на променливите σ, за който $\phi(\sigma) = 1$.

Остава да покажем, че конструирането на ϕ става за време, ограничено от някакъв полином на $n = |\alpha|$. За целта да отбележим, че общият брой букви, които участват в КНФ на ϕ, е $pq^2+p^3+p^2x^2+p+3+3p^2x+12p^2qx = p^3+(x^2+3q+12qx)p^2+2p+3$, където с p, q и x сме означили за по-кратко $p(n)$ и константите $r+1$ и $s+1$, съответно. Полученото е очевидно полином на n. Може да се посочи константа c такава, че за построяването на всяка от буквите на КНФ (заедно със съседните ѝ знаци) са необходими не повече от c стъпки на алгоритъма. И така построеният свеждащ алгоритъм е полиномиален и следователно задачата Удовлетворимост на БФ е $\mathcal{NP}$-пълна. $\square$

6. Заключение. След като имаме подходяща $\mathcal{NP}$-пълна задача f, с използването на Лема 5.3 можем да докажем $\mathcal{NP}$-пълнотата на друга задача $g \in \mathcal{NP}$, като сведем полиномиално f към g. Първата задача, $\mathcal{NP}$-пълнотата на която е доказана чрез свеждане към нея на Удовлетворимост на БФ, е 3-Удовлетворимост на БФ, при която във всяка елементарна дизюнкция на зададената КНФ има не повече от 3 променливи. Чрез такова свеждане е доказана принадлежността на стотици интересни за теоретичната и приложната информатика задачи към класа $\mathcal{NP}$-с (виж [4]).

Голямото предизвикателство на тази теория се състои в това, че всички $\mathcal{NP}$-пълни задачи са полиномиално сводими една към друга, т.е. ако за една от тях бъде намерен полиномиален алгоритъм, такъв алгоритъм има за всяка друга задача от класа. Тогава ще се окаже, че $\mathcal{NP} = \mathcal{NP}$-$c$ и голям брой важни за практиката задачи ще могат да се решават за полиномиално време, повече или по-малко ефективно, в зависимост от бързодействието на свеждащия алгоритъм. Ако за една от тези задачи се докаже, че не съществува полиномиален алгоритъм, т.е. $\mathcal{NP} \neq \mathcal{NP}$-$c$, тогава за никоя задача от класа няма такъв алгоритъм и за решаването на такива задачи трябва да се търсят качествено нови подходи – многопроцесорни компютри с паралелни изчисления или принципно различни компютърни архитектури (квантови компютри!?). **Много опити за доказателство на едната от двете алтернативи**

са направени, но нито едно от тях не е признато за достоверно. Ако трябва да спомена какво може да се прави с $\mathcal{NP}$-пълнте задачи в рамките на традиционното използване на компютри, то има няколко възможности. Най-естествено е, да се опитаме да ограничим домена на една задача, като изберем **подходящ частен случай**, за който да опитаме да построим полиномиален алгоритъм, използвайки спецификата на избрания поддомен. Ако задачата е оптимизационна, тогава може да се опитаме да създадем **полиномиален апроксимиращ алгоритъм**, който дори да не намира оптимума, да гарантира намирането на резултат с неголямо отклонение от него. Ако нито една от двете алтернативи не е възможна, тогава вероятно задачата може да се окаже удобна за **създаване на криптосистема**, тъй като някои от най-популярните криптосистеми са базирани на сложността на $\mathcal{NP}$-пълни задачи.

ЛИТЕРАТУРА

[1] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, **2** (1956), 113–124.

[2] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, **58** (1936), 345–363.

[3] S. A. Cook. The complexity of theorem proving procedures. Proceedings of the Third Annual ACM Symposium on Theory of Computing, 1971, 151–158, doi:10.1145/800157.805047.

[4] M. R. Garey, D. S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness (ed. W. H. Freeman), 1979, ISBN 0-7167-1045-5.

[5] S. C. Kleene. λ-definability and recursiveness. *Duke Mathematical Journal*, **2**, No 2 (1936), 340–352, doi:10.1215/s0012-7094-36-00227-2.

[6] E. Post. Finite Combinatory Processes – Formulation 1. *Journal of Symbolic Logic*, **1** (1936), 103–105.

[7] A. M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, **42** (1936), 230–265. DOI:10.1112/plms/s2-42.1.230.

[8] Л. А. Левин. Универсальные задачи перебора. *Проблемы передачи информации*, **9**, No 3 (1973), 115–116.

[9] А. А. Марков. Теория алгорифмов. Москва, Изд. АН СССР, 1954.

Красимир Манев
Департамент Информатика
Нов български университет
бул. Монтевидео 20
1164 София, България
e-mail: kmanev@nbu.bg

50 YEAR SINCE PUBLISHING COOK'S THEOREM

Krassimir Manev

Algorithms are the essence of Informatics and of some classical parts of Mathematics too. But until the 1930s algorithms were considered only as a tool for solving important for science and practice tasks. Since the term **is not formal**, only from this moment the algorithms are an object of study that raised a number of important issues. For example, can the concept of algorithm be formalized? What tasks can be solved algorithmically and are there any tasks that cannot be solved with an algorithm? Why do different algorithms for the same task work for a significantly different time? Therefore, we can safely say that the foundations of the Mathematical Theory of Algorithms (which is the subject of this article) were laid in the 1930s. The year 2021 marks the 50th anniversary of the publication of a very important result of the theory and the exposition in this article focuses on this result and the following challenge that has not received an answer yet.