NP-potpuni problemi

Petrović, Josip

Master's thesis / Diplomski rad

2021

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Department of Mathematics / Sveučilište Josipa Jurja Strossmayera u Osijeku, Odjel za matematiku**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:126:909823

Sveučilište J. J. Strossmayera u Osijeku
Odjel za matematiku

Josip Petrović

NP-potpuni problemi

Diplomski rad

Osijek, 2021.

Sveučilište J. J. Strossmayera u Osijeku
Odjel za matematiku

Josip Petrović

NP-potpuni problemi

Diplomski rad

Mentor: izv. prof. dr. sc. Zoran Tomljanović

Osijek, 2021.

Sažetak

Ovaj rad se bavi teorijom računarstva. Najprije se upoznajemo s jednostavnijim modelima računanja: konačnim i potisnim automatom. Nakon toga definiramo Turingov stroj koji predstavlja opći model računanja kao i njegovu nedeterminističku verziju. Zatim definiramo vremensku složenost i upoznajemo se sa klasama P i NP te uz pomoć njih i vremenski polinomijalne redukcije definiramo NP-potpunost. Centralni dio ovog rada se odnosi na dokaz da je problem ispunjivosti (SAT) NP-potpun, a na kraju ćemo još spomenuti neke zanimljive probleme koji su također NP-potpuni.

Ključne riječi

Turingov stroj, Nedeterministički Turingov stroj, Klasa P, Klasa NP, Problem ispunjivosti (SAT), NP-potpunost, Polinomijalna redukcija, Cook-Levin teorem

Abstract

This paper deals with the theory of computation. Firstly, we introduce simpler computational models: finite and pushdown automata. Then we define a Turing machine which represents a general model of computation as well as its nondeterministic counterpart. After that we define time complexity and introduce classes P and NP. Using these classes together with polynomial time reducibility we define the NP-completeness. The central point of this paper is to prove that the satisfiability problem (SAT) is NP-complete. Lastly, we will mention a few more interesting problems that belong to the NP-complete class as well.

Key words

Turing machine, Nondeterministic Turing machine, Class P, Class NP, Satisfiability problem (SAT), NP-completeness, Polinomial time reducibility, Cook-Levin theorem

Sadržaj

1. Uvod

Računala su u modernom svijetu odavno postala mnogo više od alata. Služe za posao, zabavu, upravljanje raznim uređajima, organizaciju, automatizaciju poslova itd. No, ako pokušamo rad računala objasniti teoretski, brzo ćemo shvatiti da to nije nimalo lak zadatak. Jasno nam je da računalo prilikom svog rada prima nekakav ulaz, izvršava zadano "računanje" i vraća nam, nadamo se, zadovoljavajući izlaz. Problem zapravo nastaje kod "računanja".

Naizgled se čini da računalo može riješiti svaki problem, ali se ispostavilo da to i nije baš tako. Naime, neki problemi su *neizračunljivi*, a prepoznavanjem takvih problema se bavi teorija izračunljivosti. Neki drugi problemi, iako izračunljivi, spadaju u skupinu takozvanih *teških* problema. U slučajima velikih ulaza, za njihovo računanje nam treba nerazumna količina vremena te se pokušavaju pronaći algoritmi pomoću kojih bi se ti problemi brže rješavali. Tim problemima se bavi teorija složenosti algoritama.

Teorija složenosti grupira probleme u dvije skupine, **P** i **NP**. Jesu li te dvije skupine ekvivalentne (**P**=**NP**) ili vrijedi da je **P** $\subset$ **NP**, centralno je pitanje teorije složenosti koje je do danas ostalo neodgovoreno. Štoviše, to je jedan od sedam milenijskih problema, za čije rješavanje matematički institut Clay nudi nagradu od milijun dolara (Izvor: [5]). Značajan pomak u dokazivanju bilo koje od te dvije tvrdnje ostvarilo je uvođenje podskupa skupine **NP**, poznatog kao **NP-potpunost**.

U ovom radu ćemo pobliže objasniti pojmove izračunljivosti i kompleksnosti, precizno ćemo definirati skupine problema **P**, **NP** i **NP-potpuni** problemi, a konačni cilj rada je dokazati **NP-potpunost** jednog posebnog problema poznatijeg kao **SAT** problem (Boolean satisfiability problem). Dokaz je tehnički zahtjevan, ali tvrdnja koju dokazuje je od nepobitne važnosti u teoriji računarstva.

Za definiranje gore navedenih pojmova morat ćemo opisati i definirati *Turingov stroj*, apstraktni opći model računanja koji je sposoban izvršiti bilo koju računalnu algoritamsku proceduru. Za Turingov stroj kažemo da je jednako *moćan* kao i svako moderno računalo. Dobio je naziv po matematičaru Alanu Turingu koji ga je prvi opisao.

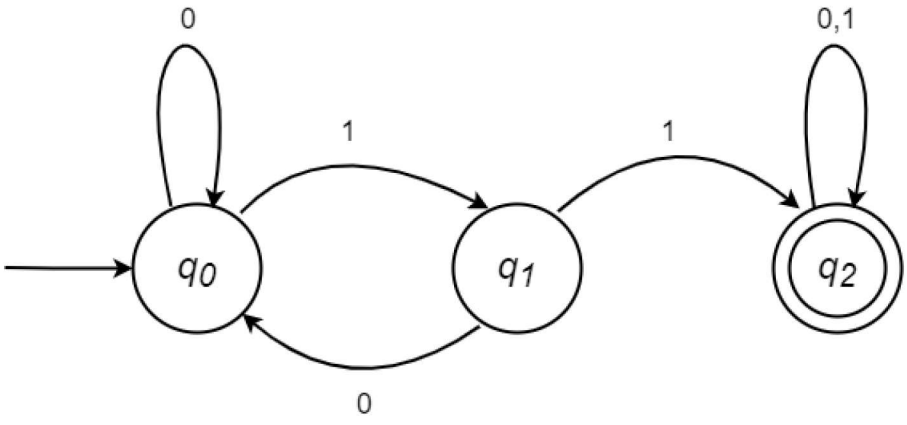
2. Složenost i automati

Postoje razni računski problemi. Neki su lagani, dok su drugi teški. Primjer laganog problema je problem sortiranja, koji se može relativno brzo rješiti. S druge strane, problem slaganja rasporeda se smatra teškim problemom. Naime, ako trebamo složiti raspored u nekoj školi tako da se 2 predmeta ne održavaju u isto vrijeme u istoj učioni, ne znamo brži način nego isprobati sve moguće kombinacije. Ali, zbog čega kažemo da je neki problem računski *težak*, dok su drugi problemi *lagani*? Ovo je najvažnije pitanje teorije složenosti, a odgovor na njega još nije ponuđen. Ipak, određeni pomaci su se napravili u tom području na način da se problemi mogu klasificirati kao lagani ili teški obzirom na njihovu računsku složenost.

Teorija automata se bavi definiranjem i svojstvima matematičkih modela računanja. Naime, teorije izračunljivosti i složenosti zahtijevaju preciznu definiciju računala. Automati su povezani s jezicima koje prihvaćaju te se uz pomoć njih i definiraju. U nastavku ćemo ukratko navesti vrste automata i grupe formalnih jezika koje prihvaćaju.

2.1. Konačni automat

Konačni automat predstavlja jednostavan model računanja, prvenstveno zato jer posjeduje vrlo limitiranu računalnu memoriju. On se sastoji od skupa stanja između kojih se kreće ovisno o *ulazu* koji automat prima. Ulaz je uglavnom prikazan u obliku *stringa*, koji predstavlja niz znakova iz nekog određenog alfabeta. Alfabet najčešće označavamo sa Σ, a primjer jednog alfabeta je $\Sigma = \{0, 1\}$. Znakovi tako definiranog alfabeta su očito 0 i 1, a stringovi koji se mogu složiti iz njega su proizvoljne permutacije konačno mnogo dostupnih znakova, npr. : 0, 1, 01, 001, 11011, 0101010 itd. Ovo inače zapisujemo kao Σ^* (u ovom slučaju može i $\{0, 1\}^*$). Specifičan skup riječi iz nekog alfabeta nazivamo jezik i najčešće označavamo velikim slovom L. Ako uzmemo ponovno alfabet $\Sigma = \{0, 1\}$, jezik može biti npr. $L = \{0, 00, 001, 1001, \ldots\}$. Jezik možemo, radi jednostavnosti, opisati i riječima. Pa tako npr. imamo jezik (iz istog alfabeta kao i prije) $L_1 = \{w | w \text{ sadrži dvije uzastopne } 1\}$. Jasno nam je da L_1 sadrži beskonačno mnogo riječi i da su one oblika: $00, 000, 100, 1100, \ldots$. Sada trebamo konstruirati konačni automat koji prihvaća takav jezik. Rješenje možemo vidjeti na slici 1.



Slika 1: Deterministički konačni automat D koji prepoznaje jezik L_1.

Deterministički konačni automat (DKA) na slici 1 sadrži 3 stanja: q_0, q_1 i q_2, pri čemu je q_0 početno stanje, a q_2 prihvaćajuće stanje. Strelice predstavljaju prijelaze između stanja i ovise o znakovima alfabeta iz kojeg je nastao jezik, u ovom slučaju $\Sigma = \{0, 1\}$. Prije nego što dobije ulaz, DKA se nalazi u početnom stanju. Zatim, DKA čita ulaz znak po znak i vrši prijelaze između stanja ovisno o znaku. Ako se, nakon što pročita posljednji znak stringa, DKA nalazi u prihvaćajućem stanju, onda DKA prihvaća dani string. DKA na slici 1 smo dali naziv D i obzirom da on prepoznaje jezik L_1, možemo reći da je L_1 jezik od D ili $L(D) = L_1$.

Uz DKA postoji i nedeterministički konačni automat (NKA). On se razlikuje od determinističkog po tome što se iz jednog stanja pomoću nekog znaka može prijeći u 0 ili više stanja pa kažemo da se računanje *grana*. Ako za neki znak ne postoji prijelaz, kažemo da grana računanja umire. Ali, ako postoji neki izbor prijelaza koji čita cijelu riječ i završava u prihvaćajućem stanju, tada NKA prihvaća ulaznu riječ. Sa ovim svojstvima, NKA se na prvu čini kao moćniji model računanja. Međutim, može se pokazati kako su DKA i NKA zapravo jednako moćni, tj. svaki DKA se može pretvoriti u ekvivalentni NKA, i obratno.

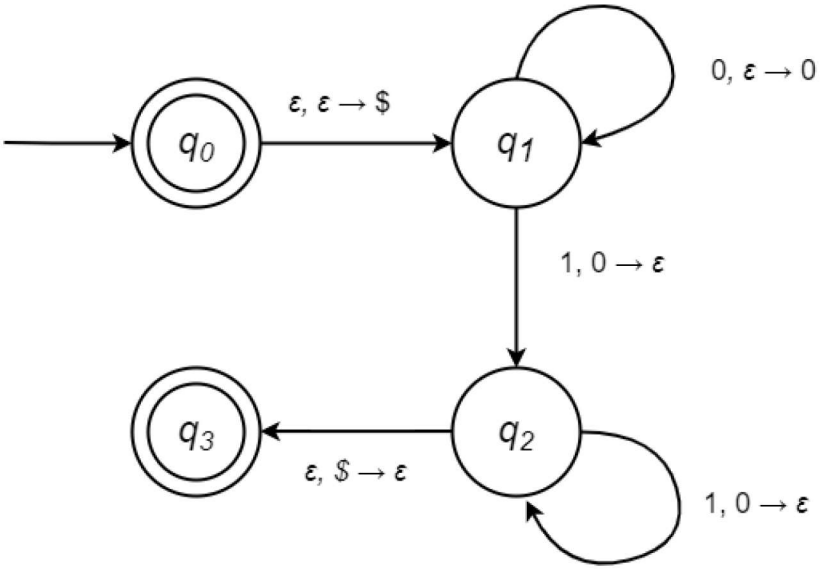
Jezik za koji možemo konstruirati DKA koji ga prepoznaje, nazivamo *regularnim jezikom*.

2.2. Potisni automat

Konačni automati imaju svoje ograničenje. Naime, oni su ograničeni na prepoznavanje regularnih jezika. Uzmimo npr. jezik $L_2 = \{0^n 1^n | n \geq 0\}$. U ovom jeziku se nalaze svi stringovi koji se sastoje od znakova 0 i 1, pri čemu se sve jedinice nalaze desno od nula i broj nula i jedinica je jednak. Pomoću leme o pumpanju za regularne jezike [2, str. 64] može se pokazati da jezik L_2 nije regularan, što znači da ne postoji KA koji ga prepoznaje. L_2 spada u skupinu tzv. *kontekstno-slobodnih jezika*.

Za prepoznavanje kontekstno-slobodnih jezika uvest ćemo *potisni automat* (PA). Potisni automat je u suštini NKA koji sadrži dodatno svojstvo: *stog*. Pomoću stoga PA može "zapamtiti" beskonačnu količinu informacija, ali za razliku od modernog računala, PA ima *last-in-first-out* pristup informacijama. To znači da PA može pročitati samo ono što je zadnje spremio u stog, a da bi dohvatio ono što se nalazi neposredno ispod vrha stoga, simbol s vrha se mora izbaciti. Skup znakova koje PA može spremiti u stog može biti različit od alfabeta koji PA čita pa taj skup nazivamo *alfabet stoga* i najčešće ga označavamo s Γ. Konstruirajmo sada PA koji prepoznaje jezik L_2.

Na slici 2 možemo vidjeti nekoliko novih elemenata. Znak ε (koristi se i kod NKA) na strelici između stanja p i q označava da PA (ili NKA) može prijeći iz stanja p u stanje q bez čitanja znakova. Znak \$ se uglavnom stavlja na dno stoga PA da bismo pri izbacivanju znakova iz stoga znali da smo došli do dna. Oznaka $a, u \to v$ označava da, kad PA pročita znak a, sa vrha stoga skida simbol u i stavlja simbol v. Dodatno, kao što i na slici 2 vidimo, dozvoljeno je stavljati, odnosno skidati simbole sa vrha stoga bez da smo neki drugi simbol skinuli odnosno stavili na vrh (oznaka ε).



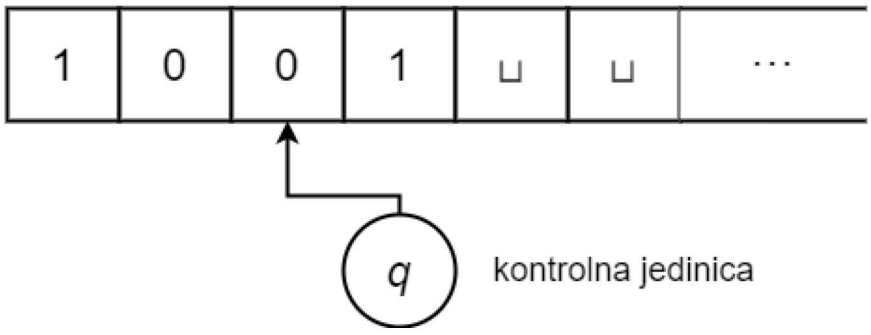
Slika 2: Potisni automat P koji prepoznaje jezik $L_2 = \{0^n 1^n | n \geq 0\}$.

Sad se postavlja pitanje, koje jezike potisni automat ne može prepoznati. Zapravo, sada je vrlo lako smisliti jezik koji nije kontekstno-slobodan. Kao što smo definirali jezik L_2, tako sada možemo definirati $L_3 = \{0^n 1^n 2^n | n \geq 0\}$. Stog nije dovoljan da PA prepozna ovako definiran jezik (što možemo i dokazati pomoću leme o pumpanju za kontekstno slobodne jezike [2, str. 206]) te ćemo morati uvesti automat koji je *moćan* koliko i svako moderno računalo, Turingov stroj.

2.3. Turingov stroj

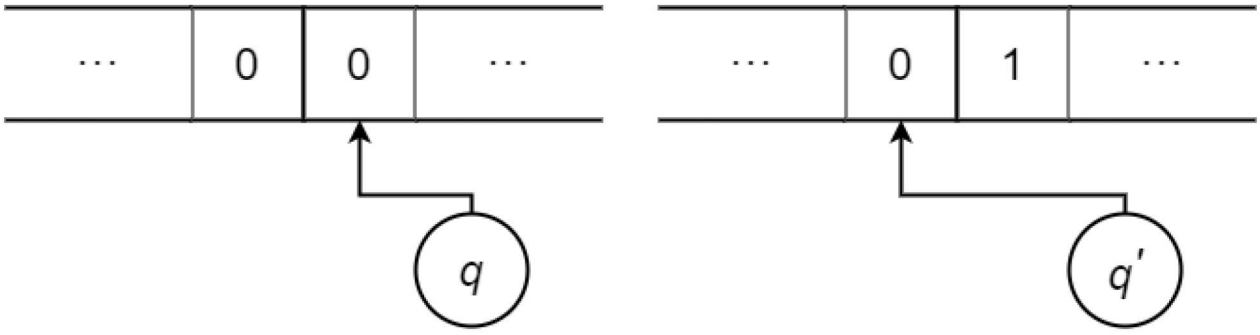
Turingov stroj (TS) nije samo jedna stepenica iznad potisnog automata. On predstavlja opći model računanja, potencijalno sposoban izvršiti bilo koji algoritam. Dobio je naziv prema engleskom matematičaru Alanu Turingu koji ga je prvi dizajnirao 30-ih godina prošloga stoljeća.

Turingov stroj se sastoji od beskonačne trake i kontrolne jedinice (glave) koja se može kretati lijevo-desno po traci te čitati simbole i pisati po njoj (slika 3).



Slika 3: Shema Turingovog stroja.

Simbol $\sqcup$ predstavlja prazan string i pretpostavit ćemo da svako polje desno od stringa (u ovom slučaju 1001) sadrži znak $\sqcup$. Korak u računanju Turingovog stroja određen je funkcijom prijelaza δ. Pretpostavimo da se TS sa slike 3 nalazi u stanju q. Kao što vidimo, kontrolna jedinica u stanju q čita znak 0. Ako želimo da TS prijeđe u stanje q' te napiše 1 umjesto 0 i pomakne se za jedno mjesto ulijevo, funkcija prijelaza će izgledati ovako: $\delta(q, 0) = (q', 1, L)$ (slika 4).



Slika 4: Tranzicija $\delta(q, 0) = (q', 1, L)$.

Ulazni string se prije početka računanja upisuje na početak trake (eventualno se u prvo polje trake upiše simbol $\sqcup$). Zatim TS računa na način da radi izmjene nad početnim stringom na zadani način. Računanje TS-a, za razliku od automata predstavljenih ranije, završava isključivo kada se TS nađe u jednom od dva stanja: prihvaćajuće ili odbijajuće. Tako dobivamo 3 mogućnosti: TS se zaustavlja u prihvaćajućem stanju, odbijajućem stanju ili računa zauvijek (ulazi u petlju). Slično kao i kod stoga potisnog automata, ovdje imamo alfabet ulaza Σ i alfabet trake Γ, pri čemu je $\Sigma \subseteq \Gamma$. Također vrijedi: $\sqcup \notin \Sigma$ i $\sqcup \in \Gamma$.

Sada ćemo uvesti formalnu definiciju Turingovog stroja:

Definicija 2.1. Turingov stroj (TS) je uređena sedmorka $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ za koju vrijedi:
1. Q je konačan skup stanja.
2. Σ je konačan ulazni alfabet koji ne sadrži simbol $\sqcup$.
3. Γ je konačan alfabet trake takav da je $\sqcup \in \Gamma$ i $\Sigma \subseteq \Gamma$.
4. $\delta : Q \times \Gamma \times \to Q \times \Gamma \times \{L, R\}$ je funkcija prijelaza.
5. $q_0 \in Q$ je početno stanje.
6. $q_{accept} \in Q$ je prihvaćajuće stanje.
7. $q_{reject} \in Q$ je odbijajuće stanje i vrijedi $q_{accept} \neq q_{reject}$.

Informacija koja predstavlja trenutnu "situaciju" u kojoj se TS nalazi nazivamo *konfiguracija*. Npr. konfiguraciju na slici 3 ćemo zapisati ovako: $10q01$. Znamo da se TS nalazi u stanju q i da čita prvu nulu slijeva te zapisujemo znak q neposredno prije prve nule. Tranziciju sa slike 4 ćemo zapisati ovako: $10q01 \implies 1q'011$ i reći da lijeva konfiguracija povlači desnu. Kažemo da TS prihvaća ulaz w ako postoje konfiguracije $C_1, C_2, \ldots, C_m$ takve da vrijedi $C_1 \implies C_2 \implies \cdots \implies C_m$ pri čemu je C_1 oblika $q_0 w$, C_m je oblika $u q_{accept} v$, a u i v su stringovi iz Γ^*. Ako je M Turingov stroj, jezik koji on prihvaća ćemo označiti sa $L(M)$.

Pretpostavimo da će Turingov stroj M stati s računanjem na svakom ulazu, tj. završiti u stanju q_{accept} ili q_{reject}. Ako takav TS prihvaća neki jezik, kažemo da M *odlučuje jezik* $L(M)$. S druge strane, ako M nije *odlučitelj* jezika $L(M)$, može se dogoditi da za neki ulaz $w \notin L(M)$ neće nikada stati s radom. U tom slučaju nećemo moći saznati pripada li dani ulaz jeziku $L(M)$ ili ne.

Definicija 2.2. Ako Turingov stroj prihvaća jezik, za taj jezik kažemo da je *Turing-prepoznatljiv* ili samo *prepoznatljiv*. Ako Turingov stroj prihvaća jezik i zaustavlja se na svakom ulazu kažemo, kažemo da je jezik *Turing-odlučiv* ili samo *odlučiv*.

Iz ove definicije može se pokazati da postoje Turing-odlučivi jezici, jezici koji su Turing-prepoznatljivi ali nisu odlučivi i jezici koji nisu čak ni Turing-prepoznatljivi. U teoriji računarstva su svi nabrojani jezici važni, ali ćemo se za potrebe ovog rada isključivo baviti odlučivim jezicima.

Sada ćemo pokazati primjer Turingovog stroja M koji odlučuje jezik $L_3 = \{0^n1^n2^n | n \geq 0\}$. Za razliku od konačnog i potisnog automata, rad TS-a ćemo opisati riječima umjesto crtanja sheme.

$M =$ "Za ulaz w:

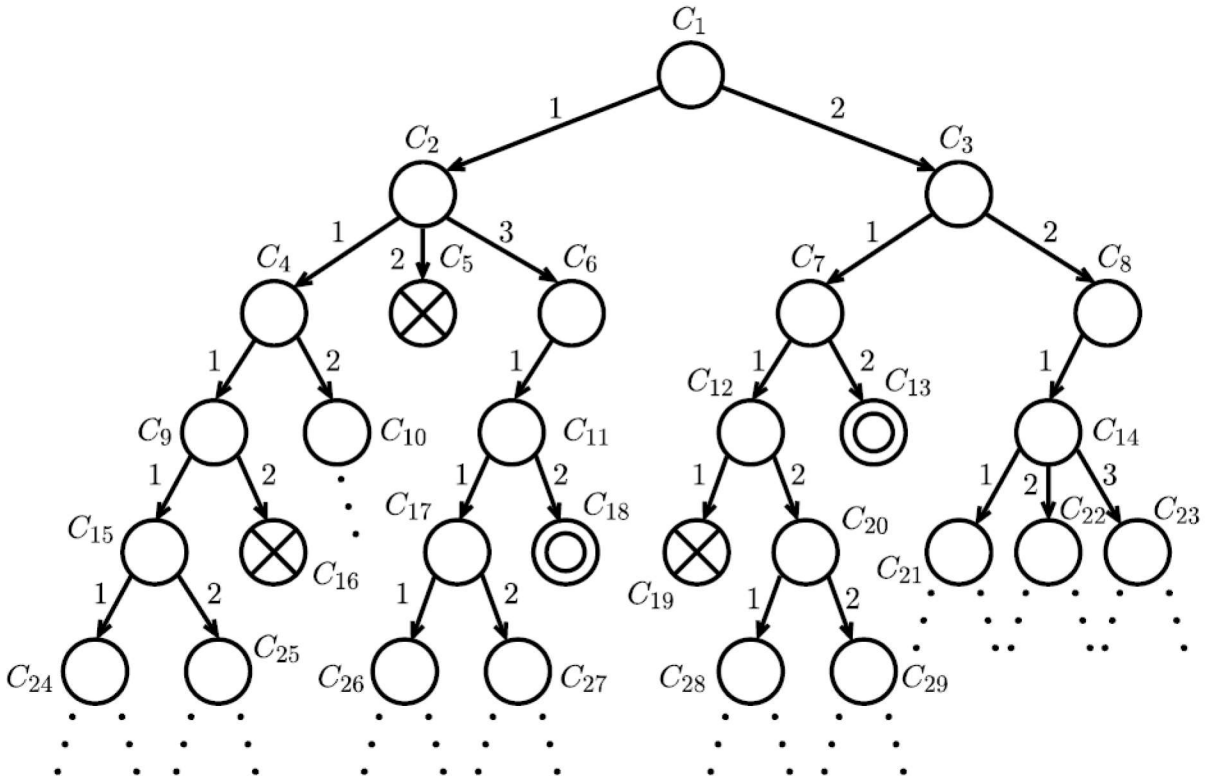
1. Skeniraj w slijeva na desno. Ako w nije oblika 0*1*2*, *odbaci*.
2. Vrati glavu na početak trake.
3. Ponavljaj sve dok više nema 0 na traci:
4. Zamijeni najljeviju 0 sa znakom x.
5. Pomiči glavu desno dok ne naiđeš na prvu 1. Ako nema znaka 1, *odbaci*.
6. Zamijeni najljeviju 1 sa znakom x.
7. Pomiči glavu desno dok ne naiđeš na prvu 2. Ako nema znaka 2, *odbaci*.
8. Zamijeni najljeviju 2 sa znakom x.
9. Vrati glavu na početak trake i idi na korak 3.
10. Ako traka sadrži znak 1 ili 2, *odbaci*. U suprotnom, *prihvati*."

2.3.1. Nedeterministički turingov stroj

Iako to nismo eksplicitno naveli, Turingov stroj koji smo dosada opisali je deterministički TS. To se da lako primjetiti jer pomoću funkcije prijelaza, za dano stanje i simbol, TS prelazi iz jedne konfiguracije u drugu konfiguraciju koja je jedinstvena. Kod *nedeterminističkog Turingovog stroja* (NTS) funkcija prijelaza izgleda malo drukčije:

$$\delta : Q \times \Gamma \times \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\}).$$

Preslikavanje od δ možemo zapisati ovako: $\delta(q, a) = \{(q_1, a_1, D_1), (q_2, a_2, D_2), \ldots, (q_m, a_m, D_m)\}$, što znači da postoji m mogućih prijelaza kada se NTS nalazi u stanju q i pročita simbol a. Kod NTS-a, svaka konfiguracija povlači 0 ili više konfiguracija. Imajući to na umu, rad danog NTS-a, ovisno i o ulazu w, možemo prikazati pomoću grafa kojemu čvorovi predstavljaju konfiguracije. Taj graf nazivamo *stablo računanja*. Primjer jednog takvog stabla možemo vidjeti na slici 5, pri čemu su prihvaćajuće konfiguracije označene sa dvostrukim krugom (kao i kod konačnog i potisnog automata), a odbijajuće imaju X unutar kruga.



Slika 5: Stablo računanja nedeterminističkog Turingovog stroja. Izvor: [3]

Za NTS M kažemo da prihvaća ulaz w ako stablo računanja koje ovisi o M i w sadrži barem jedno prihvaćajuće stanje. Kao i kod konačnog automata, NTS nam se na prvu čini moćniji od deterministčkog TS-a (u nastavku će kratica TS predstavljati deterministički Turingov stroj). Međutim, sljedeći teorem nam to opovrgava:

Teorem 2.1. *Svaki nedeterministički Turingov stroj ima ekvivalentan deterministički Turingov stroj.*

Dokaz ovog teorema se može pronaći u [1, str. 341], a ovdje ćemo samo ukratko dati ideju. Naime, po uzoru na stablo odlučivanja možemo konstruirati TS koji "pretražuje" konfiguracije stabla u širinu (Breadth-first search). Na primjer, ako pogledamo sliku 5 vidimo da je početna konfiguracija C_1. TS ne može istovremeno ući u konfiguracije C_2 i C_3 kao NTS, ali zato može ući u C_2, provjeriti je li ona prihvaćajuća, i ako nije, vratiti se u C_1 i tek tada ući u C_3. Ako ni C_3 nije prihvaćajuća konfiguracija, TS se vraća u C_2 i iz njega provjerava konfiguracije C_4, C_5 i C_6. Ako nastavi raditi na taj način, ovaj konkretni TS će doći do prihvaćajuće konfiguracije (u ovom slučaju C_{13}).

Iako sada znamo da NTS nije moćniji od TS-a, iz prethodnog primjera možemo primjetiti da računanje koje TS provodi da bi imitirao rad ekvivalentnog NTS-a prođe kroz znatno veći broj konfiguracija dok ne pronađe onu prihvaćajuću. Mi do sada nismo uvodili pojam "vremena" jer su svi automati dosad opisani u radu zapravo apstraktni objekti. U nastavku ćemo uvesti pojam vremenske složenosti gdje će se pokazati prednost nedeterminizma nad determinizmom.

2.4. Vremenska složenost

Definicija 2.3. Neka je M deterministički TS koji staje na svim ulazima. *Vremenska složenost* od M je funkcija $f : \mathbb{N} \to \mathbb{N}$ gdje $f(n)$ predstavlja maksimalan broj koraka koji M prolazi prije zaustavljanja za neki ulaz duljine n. U tom slučaju ćemo reći da M računa u vremenu $f(n)$ i da je M *$f(n)$-vremenski Turingov stroj.*

Sada ćemo u priču uvesti i takozvanu *veliko O notaciju* (eng. Big-O notation). Treba naglasiti da se ovdje radi o notaciji koja omeđuje vremensku složenost odozgo (eng. *worst-case*) te da postoje i druge notacije. Započet ćemo s definicijom:

Definicija 2.4. Neka su f i g funkcije $f, g : \mathbb{N} \to \mathbb{R}^+$. Kažemo da je $f(n) = O(g(n))$ ako postoje $c, n_0 \in \mathbb{N}$ takvi da $\forall n \geq n_0$ vrijedi $f(n) \leq cg(n)$.

Ova notacija nam prvenstveno služi za pojednostavljivanje vremenske složenosti koju smo maloprije definirali. To radi na način da za neku specifičnu vremensku složenost (funkciju) uzima u obzir samo najdominantniji član. Na primjer, ako imamo vremensku složenost $f(n) = 2n^3 + 3n^2 + 4n + 5$, reći ćemo da je $f(n) = O(n^3)$ (možemo uzeti npr. $c = 3$ i $n_0 = 100$). Ako neki problem, radi praktičnosti, podijelimo na podprobleme i računamo posebno vremensku složenost svakog od njih, ukupnu vremensku složenost dobivamo tako da zbrojimo vremenske složenosti podproblema. Na primjer, u nekom problemu smo dobili sljedeće vremenske složenosti podproblema: $O(n^2), O(n), O(n)$ i $O(1)$ (konstantno vrijeme). Ukupno vrijeme je tada $O(n^2) + O(n) + O(n) + O(1) = O(n^2)$.

Definicija 2.5. Neka je t funkcija $t : \mathbb{N} \to \mathbb{R}^+$. *Vremenska klasa složenosti* TIME($t(n)$) se definira kao

$$\text{TIME}(t(n)) = \{L | \text{ postoji deterministički TS koji odlučuje jezik L u vremenu } O(t(n))\}.$$

Problem pretraživanja niza duljine n pripada klasi TIME(n), dok npr. problem pretraživanja "Bubble sort" pripada klasi TIME(n^2). Po nazivu primjećujemo da vremenska klasa složenosti služi da smjestimo više jezika (problema) u jednu klasu iako ti problemi međusobno mogu biti vrlo različiti.

2.5. Klase P i NP

U ovom odjeljku priču o vremenskoj složenosti podižemo na novu razinu. Iako nam na prvu ne bi palo napamet izjednačiti 2 jezika čije su vremenske složenosti npr. $O(n^2)$ i $O(n^4)$, ovdje ćemo učiniti upravo to. Dapače, izjednačit ćemo sve jezike koji su odlučivi u vremenenu $O(n^k)$, za bilo koji $k \in \mathbb{N}$. Ta klasa će predstavljati probleme koji su rješivi na stvarnom računalu u neko razumno vrijeme. Naravno da $O(n^{1000})$ ne predstavlja razumno vrijeme prema našem poimanju, ali se u praksi pokazalo da, kada se problem može rješiti u vremenu $O(n^k)$, konstanta k se često može svesti na prilično malu. Iz ovoga nam slijedi definicija klase **P**:

Definicija 2.6. **P** je klasa jezika koji su odlučivi u polinomijalnom vremenu na determinističkom Turingovom stroju, tj.

$$\mathbf{P} = \bigcup_{k \in \mathbb{N}} \text{TIME}(O(n^k))$$

Uzmimo za primjer problem dohvatljivosti (eng. reachability). Neka je G usmjereni graf koji sadrži m vrhova. Za vrhove s i t potrebno je odrediti postoji li usmjereni put između s i t. Prvo ćemo problem pokušati rješiti *brute-force* algoritmom. Algoritam uzima u obzir sve potencijalne puteve od s do t oblika $s = v_0, v_1, v_2, \ldots, v_{i-1}, v_i = t$. Broj puteva koji algoritam mora provjeriti je

$$1 + m + m^2 + \ldots + m^{m-2},$$

što ćemo u Big-O notaciji zapisati kao $O(m^{m-2})$, a to je eksponencijalno obzirom na broj vrhova u grafu. Pokušat ćemo ponovo sa drukčijim pristupom. Treba samo napomenuti da ćemo u sljedećem algoritmu (i dalje u radu) prilagoditi ulaz na način da objekti koje koristimo (u ovom slučaju graf i vrhovi) budu zapisani u obliku stringa. Nećemo uvijek eksplicitno objašnjavati kako to radimo, nego ćemo samo upisati objekte unutar znakova $\langle\rangle$, a to podrazumijeva da promatrane objekte možemo razumno kodirati. Na primjer, graf možemo kodirati tako da prezentiramo liste njegovih vrhova i bridova u obliku stringa. Algoritam glasi:

$M =$ "Na ulazu $\langle G, s, t\rangle$ gdje je G usmjereni graf, a s i t su vrhovi:
1. Neka su R i S skupovi koji sadrže vrh s.
2. Sve dok $S \neq \emptyset$, učini sljedeće za svaki v iz S:
3. Za svaki brid (v, v'), pri čemu $v' \notin R$, učini sljedeće:
4. $R \leftarrow R \cup \{v'\}$.
5. $S \leftarrow S \cup \{v'\}$
6. $S \leftarrow S \backslash \{v\}$
7. Ako je $t \in R$, *prihvati*. U suprotnom, *odbaci*."

Umjesto da provjerava sve potencijalne puteve, ovaj algoritam koristi skup u kojemu se na početku nalazi samo vrh s, a zatim u njega dodaje vrhove koji su bridom povezani sa nekim vrhom iz skupa. Na kraju provjerava nalazi li se vrh t unutar skupa.

Analizom vremenske složenosti ovog algoritma primjećujemo da na nju glavni utjecaj imaju petlje pod točkama 2. i 3. koje rade u vremenima $O(m)$, odnosno $O(m^2)$. Tako nam je ukupno vrijeme izvršavanja algoritma $O(m \times m^2)$, čime smo pokazali da problem dohvatljivosti spada u klasu **P**. Za probleme iz klase **P** kažemo da ih možemo "brzo riješiti".

Prisjetimo se nedeterminističkog Turingovog stroja i načina na koji on računa (slika 5). Rekli smo da da NTS N prihvaća ulaz w ako postoji prihvaćajuća konfiguracija u stablu računanja ovisnom o N i w. Ako takva konfiguracija postoji, vrijeme računanja koje je NTS-u potrebno da dođe od početne konfiguracije do te prihvaćajuće je ništa drugo nego broj koraka koji NTS koristi da dođe od jedne do druge na toj grani računanja, ne uzimajući u obzir ostale grane. Ako npr. pogledamo ponovno sliku 5, primjetit ćemo da NTS dolazi do konfiguracije C_{13} u samo 3 koraka ($C_1 \implies C_3 \implies C_7 \implies C_{13}$). Vremenska složenost NTS-a je pak broj koraka potreban NTS-u da dođe do korijena najduže grane računanja. To ćemo sada formalno definirati:

Definicija 2.7. Neka je N NTS koji odlučuje neki jezik. Vremenska složenost od N je funkcija $f : \mathbb{N} \to \mathbb{N}$, gdje je $f(n)$ maksimalni broj koraka koji N koristi na bilo kojoj grani računanja na proizvoljnom ulazu duljine n.

Vratimo se nakratko na problem dohvatljivosti i brute-force algoritam u kojem smo isprobavali sve potencijalne puteve od vrha s do vrha t. Vidjeli smo da ćemo u najgorem slučaju

isprobati $O(m^{m-2})$ koraka koristeći deterministički TS. Kod nedeterminističkog pristupa problemu sve potencijalne puteve provjeravamo istovremeno. U tom slučaju ćemo završiti sa $O(m^{m-2})$ grane računanja, ali u kontekstu vremenske složenosti to je nevažna činjenica. Obzirom da smo pretpostavili da TS provjerava svaki od potencijalnih puteva u konstatnom vremenu, vremenska složenost NTS-a će nam tada biti $O(m)$.
Ovim primjerom smo vidjeli prednost NTS-a nad TS-om. Iako su jednako moćni po pitanju prepoznavanja i odlučivanja jezika, vremenska složenost ipak daje znatnu prednost nedeterminizmu. O tome nam govori i sljedeći teorem:

Teorem 2.2. *Neka je $t(n)$ funkcija gdje je $t(n) \geq n$. Tada svaki $t(n)$ vremenski nedeterministički Turingov stroj ima ekvivalentni $2^{O(t(n))}$ vremenski deterministički Turingov stroj.*

Dokaz teorema se može pronaći u [4, str. 284]. Promotrimo sada drugi problem. *Hamiltonov put* u grafu je put koji prolazi svim vrhovima točno jednom. Graf koji sadrži Hamiltonov put zove se Hamiltonov graf. Kod problema dohvatljivosti uspjeli smo konstruirati algoritam koji ga brzo rješava. Za problem Hamiltonovog puta brzi algoritam još uvijek nije pronađen, a ne zna se ni da li postoji. Ako Hamiltonov put postoji u grafu sa m vrhova možemo ga zapisati u obliku niza kojim put prolazi, tj. $v_1, v_2, \ldots, v_m$. Brute-force algoritam bi u pronalaženju Hamiltonovog puta trebao isprobati sve permutacije vrhova grafa pa bi vrijeme njegovog izvršavanja bilo $O(m!)$. Postoje nešto brži algoritmi, ali ni jedan ne rješava problem u polinomijalnom vremenu. To znači da problem Hamiltonovog puta ne možemo svrstati u klasu **P**.

U pomoć nam opet može uskočiti nedeterminiziam. NTS će nedeterministički izabrati niz vrhova u grafu i provjeriti postoji li brid između svaka dva susjedna vrha u tome nizu (i je li pravilno orjentiran i slučaju da se radi o usmjerenom grafu). Dakle, NTS problem rješava u polinomijalnom vremenu i tako uvodimo novu klasu problema:

Definicija 2.8.

$\text{NTIME}(t(n)) = \{L|$ postoji nedeterministički TS koji odlučuje jezik L u vremenu $O(t(n))\}$.

Definicija 2.9. NP je klasa jezika koji su odlučivi u polinomijalnom vremenu na nedeterminističkom Turingovom stroju, tj.

$$\mathbf{NP} = \bigcup_{k \in \mathbb{N}} \text{NTIME}(O(n^k))$$

Na primjer, problem Hamiltonovog puta možemo riješiti uz pomoć NTS-a N na način da N odabere redoslijed vrhova u grafu nedeterministički, a zatim provjerava jesu li svaka dva uzastopna vrha povezana bridom. Takvim pristupom NTS odlučuje problem Hamiltonovog puta u polinomijalnom vremenu (točnije, u vremenu $O(m)$) pa zaključujemo da problem Hamiltonovog puta pripada klasi **NP**.

2.6. Problem ispunjivosti (SAT)

Posebno važan problem u ovome radu nam je takozvani *problem ispunjivosti* (eng. *satisfiability problem*) ili skraćeno, SAT. Za definiranje SAT-a bit će nam potrebni sljedeći elementi:

1. *Booleove varijable*, tj. varijable čije vrijednosti mogu biti TRUE ili FALSE (1 ili 0).
2. Binarni Booleovi operatori $\wedge$ i $\vee$ koje predstavljaju logičko I, odnosno ILI.
3. Unarni Booleov operator $\neg$ koji predstavlja logičku negaciju. Radi jednostavnosti, negaciju varijable x ćemo zapisivati $\overline{x}$ umjesto $\neg x$.
4. Zagrade koje grupiraju varijable i operatore radi mijenjanja zadanog prioriteta operatora: $\neg > \wedge > \vee$.

x_1	$\overline{x_1}$	x_2	$x_1 \wedge x_2$	$x_1 \vee x_2$
0	1	0	0	0
0	1	1	0	1
1	0	0	0	1
1	0	1	1	1

Tablica 1: Tablica djelovanja logičkih operatora.

Varijable koje se pojavljuju u problemu SAT ćemo označavati sa $x_1, x_2, \ldots x_k$, gdje je $k \geq 1$. Bilo da se varijabla x pojavljuje u izvornom ili negiranom obliku, to ćemo nazivati *literal*. *Booleova formula*, koju ćemo označavati sa ϕ, je izraz koji se sastoji od niza literala međusobno odvojenih operatorima $\wedge$ ili $\vee$, pri čemu ti literali mogu biti grupirani unutar zagrada, kao npr.:

$$\phi_1 = x_1 \wedge (x_2 \wedge x_3) \wedge (\overline{x_2} \vee x_1) \vee \overline{x_3}.$$

Reći ćemo da je formula u *konjuktivnoj normalnoj formi* (CNF) ako se sastoji od konjukcije *klauzula* $C_1, C_2, \ldots C_c$, $c \in \mathbb{N}$, gdje klauzula predstavlja disjunkciju literala grupiranih unutar zagrada. Primjer Booleove formule u CNF:

$$\phi_2 = x_1 \wedge (x_2 \vee \overline{x_1}) \wedge (\overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee x_4).$$

Kažemo da je Booleova formula *ispunjiva* ako postoji neka dodjela nula i jedinica varijablama tako da vrijednost ukupne formule bude 1. Iz toga definiramo SAT:

$$SAT = \{\langle\phi\rangle | \phi \text{ je ispunjiva formula}\}.$$

Na primjer, ako u formuli ϕ_2 dodjelimo varijablama x_1, x_2 i x_4 vrijednost 1, a varijabli x_3 vrijednost 0, dobit ćemo:

$$\phi_2(1, 1, 0, 1) = 1 \wedge (1 \vee 0) \wedge (0 \vee 1) \wedge (0 \vee 1) = 1.$$

To znači da je $\phi_2 \in SAT$.

Za problem ispunjivosti još uvijek nije pronađen *brzi* način određivanja je li formula ispunjiva. Možemo jedino isprobati sve moguće dodjele vrijednosti varijablama što nas dovodi u vremensku složenost $O(2^m)$, gdje je m broj varijabli u Booleovoj formuli. NTS s druge strane

dodjeljuje vrijednosti varijablama nedeterministički te samo treba provjeriti je li ukupna vrijednost formule pri takvoj dodjeli vrijednosti varijablama jednaka 1. To znači da NTS odlučuje SAT u vremenu $O(m)$ što pokazuje da problem ispunjivosti pripada klasi **NP**. Važnost problema ispunjivosti pokazat će se u idućem poglavlju.

Jedan od najvećih neriješenih problema u teoriji računarstva je problem "**P** versus **NP**". Naime, postoje dvije mogućnosti odnosa tih skupova: **P**=**NP** i **P** $\subset$ **NP**. Iako se druga mogućnost čini logičnija, možemo vidjeti u primjeru problema dohvatljivosti (str. 9) da onda kada smo pronašli algoritam koji problem rješava u polinomijalnom vremenu, taj problem svrstavamo u skup **P**. Isto tako, kada bismo pronašli algoritme koji brzo rješavaju sve probleme iz **NP**, dobili bi izjednačenje ta dva skupa. Srećom, ne moramo tražiti brzi algoritam za svaki od tih problema posebno, već će biti dovoljno (ako je moguće) pronaći brzo rješenje za samo jedan problem iz specifinčnog skupa problema koji je podskup skupa **NP**, što ćemo vidjeti u nastavku.

3. NP-potpunost

Kada smo se upoznavali sa Turingovim strojem govorili smo o njemu kao prepoznavatelju ili odlučitelju nekog jezika. Sada ćemo uvesti definiciju koja govori o izračunljivosti funkcija uz pomoć TS-a:

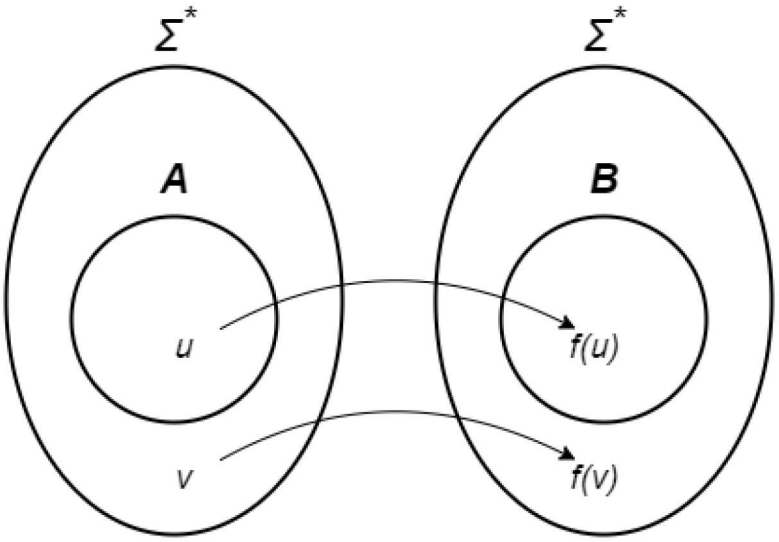
Definicija 3.1. Kažemo da je funkcija $f : \Sigma^* \to \Sigma^*$ *izračunljiva* ako postoji Turingov stroj koji za svaki ulaz $w \in \Sigma^*$ staje sa stringom $f(w)$ zapisanim na traci.

Izračunljive funkcije nam između ostalog služe da pretvorimo jedan problem u drugi. Naime, ako imamo nekakva dva problema (jezika) A i B, možemo pokazati da se rješavanjem jednog problema može pronaći rješenje onog drugog. Ugrubo govoreći, relacija $A \leq B$ govori da se problem A može riješiti pomoću rješenja problema B. Na primjer, neka imamo nekakav ulaz w za koji želimo pronaći odlučitelja koji će nam odgovoriti vrijedi li $w \in A$ ili $w \notin A$. Ako vrijedi $A \leq B$, onda možemo ulaz w modificirati na neki način da dobijemo string w' za koji znamo (uz pomoć odlučitelja za B) da vrijedi $w' \in B$ ili $w' \notin B$. Tada jednostavno možemo uzeti rješenje dobiveno za ulaz w' i primjeniti ga na ulaz w. U tom slučaju kažemo da je problem A reducibilan na problem B, a modifikacija stringa w u string w' se može gledati kao djelovanje funkcije f, tj. $f(w) = w'$. Ako je redukcija izračunljiva nekim Turingovim strojem, onda se ona naziva *redukcija mapiranjem* i označava sa $A \leq_m B$. Iz toga nam slijedi definicija:

Definicija 3.2. Jezik $A \subset \Sigma^*$ je reducibilan mapiranjem na jezik B (i pišemo $A \leq_m B$) ako postoji izračunljiva funkcija $f : \Sigma^* \to \Sigma^*$ takva da za svaki w vrijedi:

$$w \in A \iff f(w) \in B.$$

f zovemo redukcijom A na B.



Slika 6: Redukcija jezika A na jezik B.

O odlučivanju problema A uz pomoć problema B govori nam sljedeći teorem:

Teorem 3.1. *Ako vrijedi $A \leq_m B$ i B je odlučiv, onda je A odlučiv.*

Dokaz se može pronaći u [3, str. 174].
Priču o redukciji problema možemo podići na jednu višu razinu. Kao što smo pokazali da se jedan problem može odlučiti uz pomoć redukcije na neki drugi, tako možemo koristi redukciju za utvrđivanje pripadnosti jezika klasi **P**.

Definicija 3.3. Kažemo da je funkcija $f : \Sigma^* \to \Sigma^*$ *vremenski polinomijalno izračunljiva* ako postoji vremenski polinomijalni Turingov stroj koji za svaki ulaz $w \in \Sigma^*$ staje sa stringom $f(w)$ zapisanim na traci.

Definicija 3.4. Jezik $A \subset \Sigma^*$ je *vremenski polinomijalno reducibilan mapiranjem* na jezik B (i pišemo $A \leq_p B$) ako postoji vremenski polinomijalno izračunljiva funkcija $f : \Sigma^* \to \Sigma^*$ takva da za svaki w vrijedi:

$$w \in A \iff f(w) \in B.$$

f zovemo redukcijom A na B.

Slično kao i za odlučivost, imamo teorem koji govori o pripadnosti klasi **P** obzirom na polinomijalno reducibilno mapiranje:

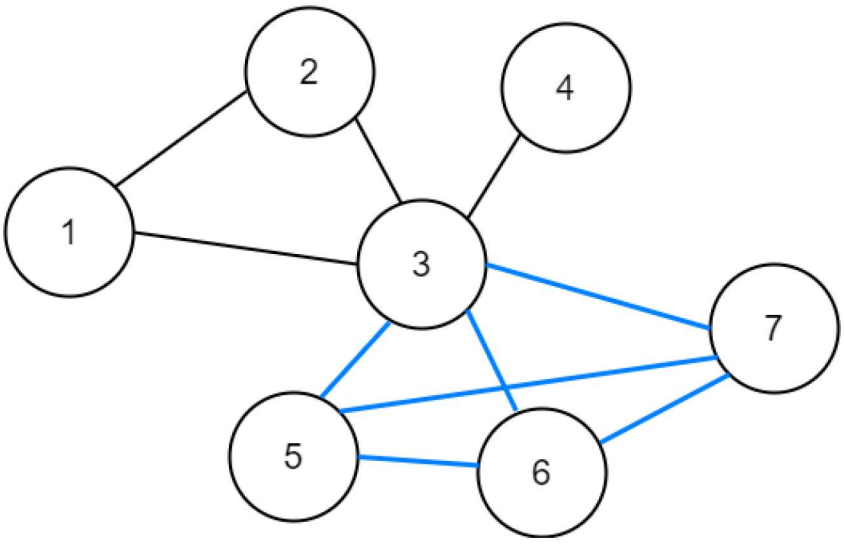
Teorem 3.2. *Ako vrijedi $A \leq_p B$ i $B \in \boldsymbol{P}$, tada je i $A \in \boldsymbol{P}$.*

Dokaz se može pronaći u [3, str. 174].
Ovaj teorem nam daje vrlo zanimljiv rezultat. Na primjer, ako imamo jezike $A, B \in \mathbf{NP}$ za koje vrijedi $A \leq_p B$ i uspijemo pronaći vremenski polinomijalni TS koji odlučuje jezik B, tada će oba jezika pripadati klasi **P**.

Neusmjereni graf u kojem su svi vrhovi međusobno povezani bridom naziva se *potpun graf*. Klika u grafu je podgraf koji je potpun graf. *K-klika* je klika koja sadrži točno k vrhova. Problem *KLIKA* znači odrediti sadrži li dani graf kliku s određenim brojem vrhova, tj.

$$KLIKA = \{\langle G, k\rangle | G \text{ je neusmjeren graf koji sadrži } k\text{-kliku}\}.$$

Problem *KLIKA* spada u skup **NP** kao i problem ispunjivosti (SAT) koji smo opisali u prošlom poglavlju.



Slika 7: Graf koji sadrži 4-kliku.

Pokazat ćemo da možemo napraviti redukciju problema SAT na problem *KLIKA* (Ova redukcija je napravljena prema [2]). Pretpostavimo da je ϕ Booleova formula u CNF, tj. oblika

$$\phi = \bigwedge_{i=1}^{k} \bigvee_{j=1}^{k_i} a_{i,j}$$

gdje k predstavlja broj klauzula u formuli, k_i je broj literala u i-toj klauzuli, a $a_{i,j}$ je literal. Trebamo opisati $f(\phi) = (G, k)$ na način da vrijedi $\phi \in SAT \iff$ graf G sadrži k-kliku. $G = (V, E)$ je neusmjeren graf, što znači da je za neka dva vrha $v_p, v_r \in V$ koja su povezana bridom svejedno pišemo li (v_p, v_r) ili (v_r, v_p) za spomenuti brid. Graf ćemo konstruirati na način da vrhovi odgovaraju literalima iz grafa, tj.

$$V = \{(i, j) | 1 \leq i \leq k \wedge 1 \leq j \leq k_i\}.$$

Vrijedit će nam da su vrhovi (i, j) i (l, m) povezani bridom ako i samo ako se odgovarajući literali nalaze u razlčitim klauzulama i ako ne vrijedi $a_{i,j} = \overline{a_{l,m}}$. Skup E možemo zapisati ovako:

$$E = \{((i, j), (l, m)) | i \neq l \wedge a_{i,j} = \overline{a_{l,m}}\}.$$

Ako je formula ϕ ispunjiva, to znači da postoji dodjela vrijednosti varijablama takva da u svakoj klauzuli barem jedan literal a_{i,j_i} ima vrijednost 1, tj. postoje literali $a_{1,j_1}, a_{2,j_2}, \ldots, a_{k,j_k}$ koji imaju vrijednost 1. Osim što se ti literali nalaze u različitim klauzulama, vrijedi i da nisu negacija jedan drugog što nam garantira da će njima odgovarajući vrhovi

$$(1, j_1), (2, j_2), \ldots (k, j_k)$$

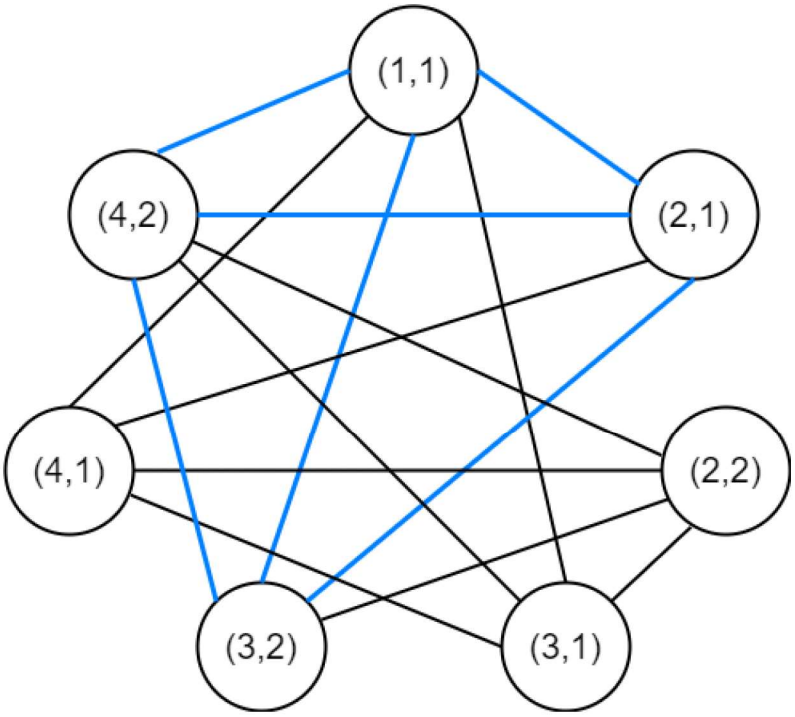
činiti potpun graf, a time i k-kliku grafa G. S druge strane, pretpostavimo da imamo graf G koji sadrži k-kliku. Obzirom da nijedan od literala koji odgovaraju tim vrhovima ne predstavlja negaciju nekog drugog literala, postoji dodjela vrijednosti koja ih sve čini istinitima. Znamo i da ti literali pripadaju različitim klauzulama, što znači da sve klauzule imaju vrijednost 1. Iz toga slijedi da nam je ϕ ispunjiva formula.

Uzmimo ponovno formulu

$$\phi_2 = x_1 \wedge (x_2 \vee \overline{x_1}) \wedge (\overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee x_4)$$

sa 4 klauzule i ukupno 7 literala. Odgovarajući graf koji ćemo konstruirati pomoću gore navedene procedure će sadržavati 7 vrhova (slika 8), a vrijedit će da taj graf sadrži 4-*kliku* akko je ϕ_2 ispunjiva formula.

Znamo otprije da je formula ϕ_2 ispunjiva za $x_1, x_2, x_4 = 1$ i $x_3 = 0$ pa prema tome zaključujemo i da graf konstruiran prema proceduri iznad sadrži 4-*kliku* što možemo i vidjeti na slici 8.



Slika 8: Graf konstruiran prema formuli ϕ_2.

Možemo reći da je ulaz $\langle\phi\rangle$ veličine n ako ϕ sadrži n literala. Da bismo konstruirali graf G morat ćemo prvo ubaciti n vrhova, a zatim ćemo te vrhove povezati bridovima prema ranije navedenim svojstvima. Maksimalan broj bridova koji možemo imati je $n(n-1)/2$ pa time vidimo da će nam se redukcija jednog problema na drugi odvijati u polinomijalnom vremenu. Napokon dolazimo do definicije **NP**-potpunog jezika.

Definicija 3.5. Jezik B je **NP**-potpun ako zadovoljava dva uvjeta:

1. $B \in \mathbf{NP}$.
2. Za svaki $A \in \mathbf{NP}$ vrijedi $A \leq_p B$.

Ova definicija nas dovodi do jednog od najvećih teorema teorije računarstva:

Teorem 3.3. *Ako je B **NP**-potpun i $B \in \boldsymbol{P}$, vrijedi $\boldsymbol{P} = \boldsymbol{NP}$.*

Dokaz teorema slijedi direktno iz definicija 3.4. i 3.5.

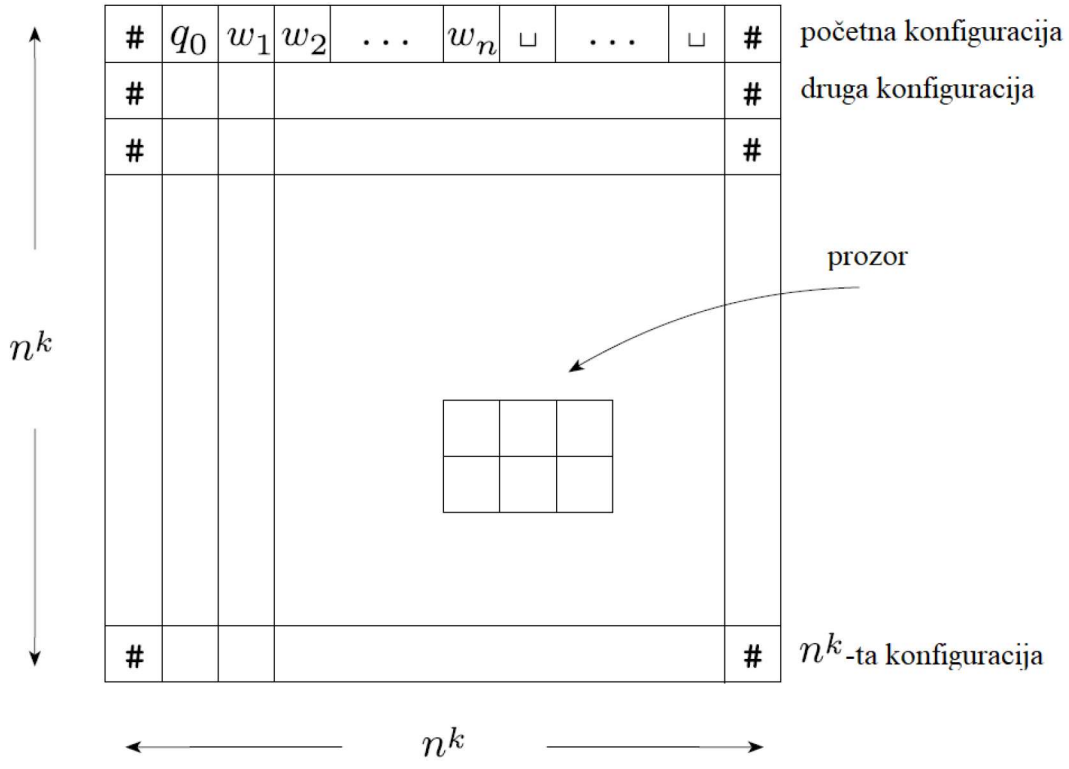
3.1. Cook-Levin teorem

Pokazali smo da se problem SAT može reducirati na problem $KLIKA$ i obratno te da su oba problema u skupu **NP**. Međutim, to nam nije dovoljno da možemo tvrditi da su oni **NP**-potpuni jer bi trebalo sve probleme iz **NP** moći reducirati na njih. Time dolazimo do centralnog dijela ovog rada:

Teorem 3.4. (Cook-Levin) *Problem SAT je **NP**-potpun.*

Koristeći [4] napravit ćemo dokaz ovog teorema. Tehnički je dosta zahtjevan pa ćemo dati ideju prije nego se upustimo u njega. Prema definiciji 3.5 bismo trebali prvo dokazati da je $SAT \in \mathbf{NP}$. Međutim, to je lakši dio dokaza te se možemo jednostavno referirati na opis koji smo dali na stranici 12. Teži dio je pokazati da se svi jezici iz skupa **NP** mogu polinomijalno reducirati na SAT. To ćemo napraviti tako da ćemo uzeti predstavnika jezika iz skupa **NP** (nazvat ćemo ga jezik A) i kreirati polinomijalnu redukciju sa A na SAT. Ta redukcija prima nekakav ulaz w i daje Booleovu formulu ϕ koja simulira djelovanje NTS-a na ulazu w. Vrijedit će da $w \in A \iff \phi$ je zadovoljiva formula.

Dokaz. Neka je N nedeterministički Turingov stroj koji odlučuje jezik $A \in \mathbf{NP}$ u vremenu n^k za neki konstantan k. *Tablica* koja prikazuje računanje stroja N na ulazu w je tablica dimenzija $n^k \times n^k$ čiji retci predstavljaju konfiguracije na nekoj grani računanja stroja N na ulazu w.



Slika 9: $n^k \times n^k$ tablica konfiguracija. Izvor: [4]

Radi praktičnosti, pretpostavit ćemo da su prvi i zadnji stupac ispunjeni znakom #. U prvom retku tablice se nalazi početna konfiguracija od N na ulazu w, a zatim se svaki red nastavlja na prethodni poštujući funkciju prijelaza od N. Tablica prihvaća ulaz w ako se u nekom njezinom redu nalazi prihvaćajuća konfiguracija. Svaka prihvaćajuća tablica od N na ulazu w odgovara jednoj prihvaćajućoj grani računanja. Zato se problem prihvaćanja ulaza w svodi na pronalaženje prihvaćajuće tablice.

Sada trebamo opisati redukciju f sa jezika A na SAT. Na ulazu w, funkcija f vraća Booleovu formulu ϕ, tj. $f(w) = \phi$. Počet ćemo sa opisom varijabli koje se nalaze u ϕ. Neka Q i Γ predstavljaju skup stanja, odnosno alfabet trake od N te neka je $C = Q \cup \Gamma \cup \{\#\}$. Za sve i, j za koje vrijedi $1 \leq i, j \leq n^k$ i za sve $s \in C$ imat ćemo varijablu $x_{i,j,s}$. U tablici ćemo polje u i-tom retku i j-tom stupcu zapisivati $polje(i, j)$ i ono sadrži neki simbol iz skupa C. Sadržaj nekog polja ćemo prikazati pomoću varijable iz ϕ. Preciznije, ako $x_{i,j,s}$ ima vrijednost 1, onda $polje(i, j)$ sadrži s.

Složit ćemo ϕ tako da ispunjiva dodjela varijabli odgovara prihvaćajućoj tablici od N na ulazu w. ϕ će zapravo biti konjukcija 4 podformule: $\phi_{polje} \wedge \phi_{start} \wedge \phi_{pomak} \wedge \phi_{prihvat}$. Svaku od njih ćemo objasniti posebno.

Kao što smo objasnili maloprije, ako bismo varijabli $x_{i,j,s}$ dodijelili vrijednost 1 to je kao kad bismo u $polje(i, j)$ smjestili simbol s. Da bi dodjela vrijednosti varijablama pravilno simulirala sadržaj svakog polja u tablici, mora se dodijeliti vrijednost 1 točno jednoj varijabli za svako polje. To znači da, ako $polje(i, j)$ sadrži neki $s \in C$, vrijednost varijable $x_{i,j,s}$ će biti 1 i vrijedit će da je $x_{i,j,s'} = 0, \forall s' \in C, s' \neq s$. Zbog toga definiramo ϕ_{polje} na sljedeći način:

$$\phi_{polje} = \bigwedge_{1 \leq i,j \leq n^k} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t \in C \\ s \neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \right) \right].$$

Primjećujemo da je ϕ_{polje} poprilično velika formula jer vanjska petlja $\bigwedge_{1 \leq i,j \leq n^k}$ prolazi svim poljima unutar tablice. Zatim za svako polje imamo dvije unutarnje petlje (unutar uglatih zagrada). Prva je disjunkcija $\bigvee_{s \in C} x_{i,j,s}$ koja prolazi svim mogućim znakovima iz skupa C. Barem jedan od njih osigurava dodjelu vrijednosti 1 varijabli koja predstavlja odgovarajuće polje. Druga petlja je konjukcija $\bigwedge_{\substack{s,t \in C \\ s \neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}})$ koja prolazi svim mogućim parovima različitih znakova iz C. Za svaki takav par znakova s i t vrijedi da najviše jedna varijabla ($x_{i,j,s}$ ili $x_{i,j,t}$) može imati vrijednost 1. Ako bi se dogodilo da obje varijable imaju vrijednost 1, onda izraz $\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}$ ima vrijednost 0, a s time i cijela formula ϕ_{polje}. Ovime smo osigurali da je dodijeljena vrijednost 1 točno jednoj varijabli za svako polje iz tablice.

Na primjer, unaprijed smo odredili da će $polje(1, 1)$ sadržavati simbol #, što znači da je $x_{1,1,\#} = 1$ i $x_{1,1,s} = 0, \forall s \in C, s \neq \#$. Obzirom da prva od dvije unutarnje petlje sadrži literal $x_{1,1,\#}$, očito je da će njena vrijednost biti 1. Unutar druge petlje će samo literal $\overline{x_{1,1,\#}}$ imati vrijednost 0, a obzirom na iteracijski uvjet $s \neq t$, taj literal nikada neće biti uparen sa samim sobom te će i ta petlja imati ukupnu vrijednost 1.

Formula ϕ_{start} je dosta jednostavnija i osigurava da je u prvom redu tablice početna konfiguracija od N na ulazu w:

$$\phi_{start} = x_{1,1,\#} \wedge x_{1,2,q_0} \wedge x_{1,3,w_1} \wedge \ldots \wedge x_{1,n+2,w_n} \wedge x_{1,n+3,\sqcup} \wedge \ldots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#}.$$

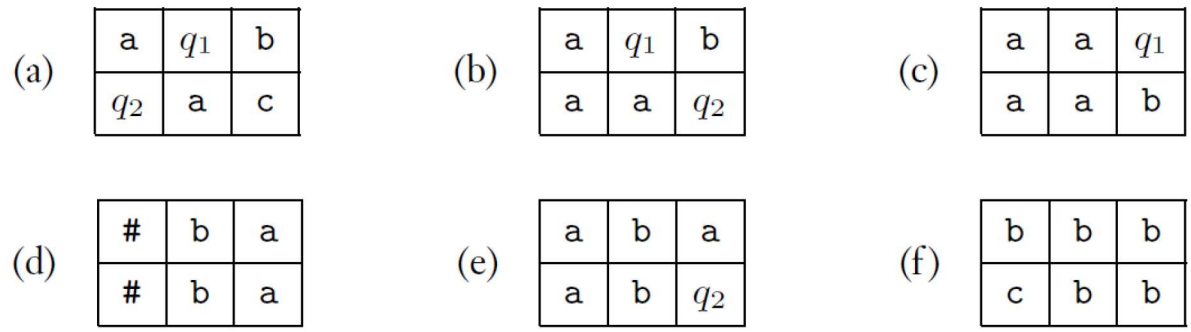
Formula $\phi_{prihvat}$ osigurava da se prihvaćajuća konfiguracija nalazi negdje u tablici:

$$\phi_{prihvat} = \bigvee_{1 \leq i,j \leq n^k} x_{i,j,q_{accept}}.$$

Posljednja formula koju trebamo opisati je formula ϕ_{pomak}. Ona će nam osiguravati da svaki redak tablice predstavlja konfiguraciju koja ispravno slijedi konfiguraciju retka iznad prema pravilima računanja od N. To radi na način da osigurava ispravnost svih *prozora* dimenzija 2×3 unutar tablice. Za prozor dimenzija 2×3 ćemo reći da je *ispravan* ako ne krši pravila funkcije prijelaza stroja N. Uzmimo za primjer da su **a**, **b** i **c** znakovi alfabeta trake, a q_1 i q_2 stanja od N. Pretpostavimo da, kad je N u stanju q_1 i pročita znak **a**, zapiše na to mjesto znak **b** i pomakne glavu udesno ostajući pritome u stanju q_1. S druge strane, ako u stanju q_1 pročita znak **b**, N može nedeterministički:

1. zapisati znak **c**, prijeći u stanje q_2 i pomaknuti se ulijevo, ili
2. zapisati znak **a**, prijeću u stanje q_2 i pomaknuti se udesno.

Formalnim zapisom to izgleda ovako: $\delta(q_1, \mathbf{a}) = \{(q_1, \mathbf{b}, \mathrm{R})\}$ i $\delta(q_1, \mathbf{b}) = \{(q_2, \mathbf{c}, \mathrm{L}), (q_2, \mathbf{a}, \mathrm{R})\}$. Na sljedećoj slici možemo vidjeti ispravne prozore koji odgovaraju prethodnom opisu:



Slika 10: Primjeri ispravnih prozora za opis iznad. Izvor: [4]

Prozori (a) i (b) na slici 10 jasno prikazuju prijelaze opisane maloprije te je lako vidjeti da su oni ispravni. U prvom retku prozora (c) N se nalazi u stanju q_1 i čita znak koji se nalazi desno od prozora. Prema opisanim pravilima, moguće da je taj znak bio **a** i da je N na to mjesto zapisao **b** i pomaknuo se udesno. Tu se ne krše nikakva pravila funkcije prijelaza, tako da je i prozor (c) ispravan. Prozor (d) ima dva identična retka što znači da je glava od N negdje desno od prozora i ne utječe na njega prijelazom iz gornjeg reda u donji te je i on ispravan. Kod prozora (e) je moguće da je glava od N u stanju q_1 neposredno desno od prozora u prvom retku i da čita znak **b**. Iz toga može prijeći u stanje q_2 i pomakuti se ulijevo te se pojaviti unutar prozora. Zbog toga je i to stanje ispravno. Zadnji primjer je prozor (f) u kojemu je moguće da je glava od N u stanju q_1 neposredno lijevo od prozora i čita znak **b** koji se u prozoru nalazi u gornjem lijevom kutu. Tu zapisuje znak **c** i pomiče se ulijevo. Ni tu se ne krše nikakva pravila prijelaza pa je i (f) ispravan.

Na sljedećoj slici možemo vidjeti neke primjere neispravnih prozora:

(a)

a	b	a
a	a	a

(b)

a	q_1	b
q_2	a	a

(c)

b	q_1	b
q_2	b	q_2

Slika 11: Primjeri neispravnih prozora. Izvor: [4]

U prozoru (a) na slici 11 promijenjen je znak **b** iako ga glava od N ne čita. Tako nešto je nedopustivo bez obzira na to kako je definirana funkcija prijelaza. U prozoru (b) N mijenja **b** u **a** i pomiče glavu ulijevo što krši pravilo funkcije prijelaza. U prozoru (c) se u donjem retku nalaze dva simbola stanja u dva različita polja.

Tvrdnja. Ako prvi redak tablice predstavlja početnu konfiguraciju i svaki 2×3 prozor tablice je ispravan, tada svaki redak predstavlja konfiguraciju koja ispravno slijedi konfiguraciju retka iznad.

Ovu tvrdnju ćemo dokazati pomoću bilo koje dvije uzastopne konfiguracije koje zbog tablice zovemo gornja i donja konfiguracija. Svako polje u tablici u kojem se ne nalazi simbol stanja niti je simbol stanja u poljima lijevo i desno do njega će biti gornje centralno polje u prozoru čiji gornji redak ne sadrži simbol stanja. Zbog toga će simbol koji se nalazi u tom polju ostati nepromijenjen u polju ispod njega, što nam osigurava da se taj isti simbol nalazi u donjoj konfiguraciji. Prozor u kojemu se simbol stanja nalazi u gornjem centralnom polju nam osigurava da se simboli u 3 polja koja se nalaze u donjem retku prozora mijenjaju u skladu s pravilima funkcije stanja. Iz ovog opisa zaključujemo da ako je gornja konfiguracija ispravna, slijedi da je i donja.

Valja napomenuti da nam je izbor dimenzija prozora važan za ovaj dokaz, tj. dimenzije 2×3 su najmanje dimenzije s kojima možemo postići željeni rezultat. To implicira da možemo i sa većim prozorima (npr. dimenzija 3×3 ili 2×4) postići isti rezultat, ali bi se previše elemenata nepotrebno provjeravalo te bi u konačnoj formuli imali više varijabli. Obzirom da pratimo dvije uzastopne konfiguracije, jasno je zašto nam je bitno da nam prva dimenzija bude 2. Vratimo se sada na primjere ispravnih prozora i pogledajmo pobliže primjer (c) (slika 10). Za taj prozor smo rekli da je ispravan iako ne znamo koji znak N čita u stanju q_1 i u koje stanje prelazi u retku ispod. Međutim, to nam ne stvara problem jer ćemo svakako provjeriti i prozor kojemu je simbol q_1 u gornjem centralnom polju i otkriti potencijalnu neispravnost. Ako bi druga dimenzija bila 2, ta neispravnost bi mogla ostati neprimjećena. Na slici 12 vidimo dva prozora dimenzija 2×2 koji bi bili ispravni prema navedenim pravilima u primjeru, ali konfiguracija u drugom retku ne bi ispravno slijedila konfiguraciju iz prvog retka. Prozor (c) je neispravan i ukazuje na pogrešku koja bi sa prozorima dimenzija 2×2 ostala neprimjećena.

(a)

a	q_1
a	b

(b)

q_1	c
b	a

(c)

a	q_1	c
a	b	a

Slika 12: Ispravni prozori dimenzija 2×2 i neispravan prozor dimenzija 2×3.

Napokon možemo konstruirati formulu ϕ_{pomak}. Ona će nam osiguravati ispravnost svih prozora tablice. Svaki prozor se sastoji od 6 polja u koja se na konačan broj načina mogu smjestiti simboli da bi prozor bio ispravan. ϕ_{pomak} nam govori da su simboli smješteni na jedan od tih načina, tj.

$$\phi_{pomak} = \bigwedge_{1\leq i<n^k, 1<j<n^k} \big((i,j)\text{-prozor je ispravan}\big).$$

(i,j)-prozor je onaj prozor kojemu je polje(i,j) u gornjoj centralnoj poziciji. Ako simbole unutar tih 6 polja označimo sa $a_1 \dots a_6$ (pri čemu je a_2 u gornjem centralnom polju), izraz "(i,j)-prozor je ispravan" možemo zamijeniti sa sljedećom formulom:

$$\bigvee_{\substack{a_1 \dots a_6 \\ \text{prozor je ispravan}}} \big(x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6}\big).$$

Prema opisanoj redukciji, formule ϕ_{polje}, ϕ_{start} i ϕ_{pomak} će za svaki ulaz od N biti ispunjive, dok će formula $\phi_{prihvat}$ biti ispunjiva ako i samo ako se prihvaćajuća konfiguracija nalazi u tablici iz čega zaključujemo da vrijedi tvrdnja $w \in A \iff \phi$ je zadovoljiva formula.
Još moramo analizirati vremensku složenost ove redukcije i pokazati da je ona polinomijalna u najgorem slučaju. Da bismo to uspjeli, trebamo ispitati veličinu formule ϕ. Tablica od N ima n^{2k} polja pri čemu u svakom polju može biti jedan od l različitih simbola, a l je broj elemenata skupa C. Međutim, l ne ovisi o veličini ulaza n, već isključivo o NTS-u N pa nam je ukupni broj varijabli $O(n^{2k})$.

Ispitat ćemo i veličinu i svake od podformula. ϕ_{polje} sadrži fiksni broj varijabli za svako polje iz tablice pa je njegova veličina $O(n^{2k})$. ϕ_{start} predstavlja prvi redak tablice te je njegova veličina $O(n^k)$. ϕ_{pomak} i $\phi_{prihvat}$ ponovno prolaze svim poljima tablice i za svako od tih polja sadrže fiksni broj varijabli (jer broj varijabli za svako polje ovisi samo o l) te je njihova veličina $O(n^{2k})$. Ukupna veličina od ϕ je tada

$$O(n^{2k}) + O(n^k) + O(n^{2k}) + O(n^{2k}) = O(n^{2k})$$

što je polinomijalno za ulaz veličine n. Svaka od podformula sadrži puno dijelova koji su gotovo identični pa zaključujemo da redukcija može generirati formulu ϕ u polinomijalnom vremenu.

□

Ovime smo dokazali da je problem SAT **NP**-potpun. Već smo ranije spominjali da je ovaj rezultat vrlo važan u području matematike i teorije računarstva jer sada znamo da, ako uspijemo pronaći brzi algoritam koji odlučuje SAT, vrijedit će $\mathbf{P} = \mathbf{NP}$. Međutim, o korisnosti tog rezultata nam govori i sljedeći teorem:

Teorem 3.5. *Ako je B **NP**-potpun i vrijedi $B \leq_p C$ za neki $C \in$ **NP**, tada je i C **NP**-potpun.*

Dokaz. Znamo da je $C \in \mathbf{NP}$ i da je svaki jezik $A \in \mathbf{NP}$ polinomijalno reducibilan na jezik B. Redukcijom sa A na B, a zatim sa B na C dobivamo redukciju A na C koja je također vremenski polinomijalna (budući da u oba slučaja imamo polinomijalnu složenost, onda je i redukcija sa A na C polinomijalna). □

Ako želimo pokazati da je neki drugi jezik **NP**-potpun, ne moramo više konstruiriati vremenski polinomijalnu redukciju sa nekakvog apstraktnog jezika $A \in \mathbf{NP}$, već možemo (ako uspijemo) konstruirati redukciju SAT-a na taj jezik.

3.2. Drugi NP-potpuni problemi

Uvest ćemo poseban oblik problema *SAT* koji nazivamo *3SAT*. Sve formule koje se u njemu nalaze su u konjuktivnoj normalnoj formi (stranica 11) i svaka klauzula u tim formulama sadrži točno 3 literala (nazivamo ih 3CNF formule), tj.

$$3SAT = \{\langle\phi\rangle | \phi \text{ je zadovoljiva } 3CNF \text{ formula.}\}.$$

Za dokazivanje tvrdnje da je neki jezik $A \in NP$-potpun obično je lakše reducirati problem *3SAT* na jezik A od problema*SAT*. Međutim, najprije bi trebali pokazati da je *3SAT* **NP**-potpun.

Korolar 3.1. *3SAT je NP-potpun.*

Dokaz. Može se pokazati da vrijedi $3SAT \in \mathbf{NP}$ slično kao i za problem *SAT* (str. 12). Da bismo pokazali da su svi problemi koji pripadaju skupu **NP** polinomijalno reducibilni na *3SAT* možemo samo izmjeniti Booleovu formulu ϕ koju u konačnici dobijemo u dokazu *Cook-Levinovog* teorema.

ϕ se već skoro nalazi u *CNF*: Formula ϕ_{polje} je velika konjukcija podformula, a svaka od tih podformula sadrži veliku disjunkciju (koja predstavlja klauzulu sa puno literala) i veliku konjukciju disjunkcija (klauzule sa 2 literala) te je ϕ_{polje} u *CNF*. Formula ϕ_{start} je velika konjukcija varijabli. Ako gledamo svaki od tih literala kao klauzulu s jednim elementom, lako vidimo i da je ϕ_{start} u *CNF*. Formula $\phi_{prihvat}$ je velika disjunkcija varijabli i predstavlja jednu klauzulu. Formula ϕ_{pomak} je velika konjukcija podformula, a svaka od tih podformula je disjunkcija konjukcija koja predstavlja sve moguće ispravne prozore. Uz pomoć distribucije, možemo disjunkciju konjukcija pretvoriti u ekvivalentnu konjukciju disjunkcija, npr.

$$(a_1 \wedge a_2) \vee (a_3 \wedge a_4) = ((a_1 \wedge a_2) \vee a_3) \wedge ((a_1 \wedge a_2) \vee a_4) = (a_1 \vee a_3) \wedge (a_2 \vee a_3) \wedge (a_1 \vee a_4) \wedge (a_2 \wedge a_4).$$

Ovakav potez može znatno povećati formulu ϕ_{pomak}, ali samo za konstantan faktor jer veličina podformula od ϕ_{pomak} ovisi isključivo o N.

Sada kad nam je cijela formula ϕ u *CNF*, trebamo je još izmjeniti tako da svaka klauzula sadrži točno 3 literala. U onim klauzulama koje imaju 1 ili 2 literala jednostavno repliciramo jednog od njih dok ne dobijemo 3 literala, npr.

$$(a_1 \vee a_2) = (a_1 \vee a_2 \vee a_2).$$

Klauzule u kojima je više od 3 literala moramo razdvojiti na više klauzula koje će sadržavati dodatne varijable zbog održanja ispunjivosti (ili neispunjivosti). Ako imamo npr. klauzulu $(a_1 \vee a_2 \vee a_3 \vee a_4)$, nju možemo razdvojiti na klauzule $(a_1 \vee a_2 \vee z) \wedge (\overline{z} \vee a_3 \vee a_4)$ gdje je z nova varijabla. Ako je originalna klauzula zadovoljiva, onda možemo dodijeliti vrijednost varijabli z tako da dvije novonastale klauzule budu obje zadovoljive. Općenito, ako klauzula sadrži l literala,

$$(a_1 \vee a_2 \vee \ldots \vee a_l),$$

možemo ju zamijeniti sa $l-2$ klauzule

$$(a_1 \vee a_2 \vee z_1) \wedge (\overline{z_1} \vee a_3 \vee z_2) \wedge (\overline{z_2} \vee a_4 \vee z_3) \wedge \ldots \wedge (\overline{z_{l-3}} \vee a_{l-1} \vee a_l).$$

Time smo dobili *3CNF* formulu koja je zadovoljiva akko je zadovoljiva i originalna formula. □

3.2.1. KLIKA

Već smo ranije pokazali da se problem *SAT* može reducirati na problem *KLIKA* u polinomijalnom vremenu. NTS odlučuje problem *KLIKA* u polinomijalnom vremenu tako što nedeterministički odabere k vrhova u grafu, a zatim provjeri jesu li svi parovi unutar tih k vrhova povezani bridom. Koristeći teorem 3.5 zaključujemo da je i *KLIKA* **NP**-potpun problem.

3.2.2. HAMILTONOV PUT

U radu smo također spominjali problem Hamiltonovog puta (str. 10) i pokazali da pripada klasi **NP**. Može se pokazati da postoji polinomijalna redukcija problema *3SAT* na problem Hamiltonovog puta. *3CNF* formula se može translatirati u graf na način da dodjela varijabli koja *3CNF* formuli daje vrijednost 1 odgovara Hamiltonovom putu u grafu. Takva redukcija nas dovodi do sljedećeg rezultata:

Teorem 3.6. *Problem Hamiltonovog puta je **NP**-potpun problem.*

Dokaz ovog teorema se može pronaći u [4, str. 314].

3.2.3. SUBSET-SUM

Problem koji nismo dosad spominjali poznat je pod nazivom *SUBSET-SUM*. Za dani skup cijelih brojeva $x_1, x_2, \ldots, x_k$ i ciljani broj t treba odrediti postoji li podskup spomenutog skupa čiji zbroj iznosi t, tj.

$$\begin{aligned} \textit{SUBSET-SUM} = \{\langle S, t\rangle | S = \{x_1, x_2 \ldots x_k\}, \text{ i za neki} \\ \{y_1, y_2, \ldots\ y_l\} \subseteq \{x_1, x_2 \ldots x_k\} \text{ vrijedi } \sum y_i = t\}. \end{aligned}$$

Na primjer, $\langle\{7, 4, 11, 8, 2, 6\}, 24\rangle \in$ *SUBSET-SUM* jer vrijedi $7 + 11 + 2 + 4 = 24$. Treba napomenuti da su $\{x_1, x_2 \ldots x_k\}$ i $\{y_1, y_2, \ldots\ y_l\}$ multiskupovi, tj. elementi se u njima mogu ponavljati.

Nije nam poznat algoritam koji odlučuje *SUBSET-SUM* u polinomijalnom vremenu. S druge strane, NTS ga može odlučiti u polinomijalnom vremenu jer odabire brojeve iz skupa S nedeterministički i zatim samo provjeri je li iznos njihovog zbroja jednak t. Takvih odabira ima konačno mnogo jer je broj kombinacija konačnog skupa također konačan. Dakle, *SUBSET-SUM* $\in$ **NP**. Sada je opet redukcijom nekog od **NP**-potpunih problema na *SUBSET-SUM* moguće pokazati da je trvdnja sljedećeg teorema istinita.

Teorem 3.7. *SUBSET-SUB je **NP**-potpun.*

U [4, str. 320] se može pronaći dokaz teorema uz pomoć redukcije problema *3SAT* na *SUBSET-SUM*.

Iz ovog poglavlja vidimo da se redukcijom vrlo često mogu povezati problemi koji pripadaju različitim područjima, a upravo na taj način smo u mogućnosti dokazivati NP-potpunost drugih problema.

Zaključak

Postoji mnogo NP-potpunih problema. Velika većina problema iz skupa NP, a koji ne pripadaju skupu P su NP-potpuni. Upravo zbog toga je razumno ponekad za problem iz NP uložiti više energije u dokazivanje da je NP-potpun nego da pripada skupu P. Pronalaženje algoritma koji rješava bilo koji od tih problema u polinomijalnom vremenu dokazuje ekvivalentnost skupova P i NP. Mi ne znamo hoće li se to i kada dogoditi, ali definitivno znamo da bi posljedice mogle biti zanimljive. Međutim, valja napomenuti da je prevladavajuće mišljenje da ta ekvivalencija ne vrijedi te da su *teški* problemi jednostavno *teški*. U cijeloj toj priči nam je posebno važan problem ispunjivosti (SAT) jer je to prvi problem za koji je dokazano da je NP-potpun, tj. da se svi problemi iz NP mogu reducirati na njega, što je ujedno i centralni dio ovog rada.

Literatura

[1] J.E. Hopcroft, R. Motwani, J.D.Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 2nd edition, 2001.

[2] J.C. Martin, *Introduction to Languages and The Theory of Computation*, McGraw-Hill Education, 4th edition, 2010.

[3] A. Maruoka, *Concise Guide to Computation Theory*, Springer, 2011.

[4] M. Sipser, *Introduction to the Theory of Computation*, Cengage Learning, 3rd edition, 2012.

[5] https://www.claymath.org/millennium-problems/millennium-prize-problems, Kolovoz 2021.