UMEÅ UNIVERSITET

$P \stackrel{?}{=} NP$ problemet

En översikt

Emma Pettersson

Emma Pettersson
Examensarbete, 15 hp
Matematik 15 hp
Ht 2018

Abstrakt

The purpose with this paper is to give an overview of the $P \stackrel{?}{=} NP$ problem and explore the impact of the three possible solutions on our computerized society. We will, inter alia, go through the $P \stackrel{?}{=} NP$ problems importance, beliefs about the problem and essential concepts.

Sammanfattning

Syftet med uppsatsen är att ge en översiktlig bild av $P \stackrel{?}{=} NP$ problemet och redogöra kring problemets tre möjliga lösningar samt vad dessa skulle innebära för vårt datoriserade samhälle. Vi kommer bland annat att gå igenom $P \stackrel{?}{=} NP$ problemets betydelse, tro kring problemet och relevanta begrepp.

Innehållsförteckning

1 Introduktion **4**

2. Begrepp och definitioner kring P $\stackrel{?}{=}$ NP **5**

2.1 Strängar och alfabeten 5
Språk 2.2 5
2.1 Turingmaskinen 6
2.1.1 Church-Turing hypotesen 7
2.2 P och NP definierade i språktermer 7
2.3 Beslutsproblem 8
2.4 Turingreduktion 9
2.5 NP-fullständiga och NP-svåra problem 10
2.5.1 SAT och 3SAT 10
2.5.2 Hamiltoncykelproblemet 12
2.6 coNP 12
2.7 NP-mellanliggande 13
2.8 Polynomiell hierarki 13
2.9 Polynomielltid 14

3 P $\stackrel{?}{=}$ NP:s betydelse **15**

3.1 Betydelse för samhället 15
3.1.1 Brute force sökningar och kryptering 15
3.1.2 P $\stackrel{?}{=}$ NP och A.I. 15
3.2 Invändningar 15
3.2.1 Den asymptotiska-invändningen 16
3.2.2 Polynomiell-tid-invändningen 16
3.2.3 Diskbänksinvändningen 17
3.2.4 Den matematiska snobberi invändningen 17
3.2.5 Surt-sa-räven-invändningen 18
3.2.6 Den självklara invändningen 18
3.2.7 Den konstruktiva invändningen 18

4 Tro kring P $\stackrel{?}{=}$ NP **19**

4.1 Anledningar till att majoriteten av forskare tror att P $\neq$ NP 19
4.2 Konsekvenser av förutfattade meningar 19
4.2.1 LBA problemet 20

5 Barriärerna **21**

5.1 Ytterligare begrepp 21
5.1.2 Orakelturingmaskin 21
5.1.3 Booleska grindar 21
5.1.3 P/poly 22

5.1.4 AC0 22
5.1.5 TC0-kretsar och MAJORITET grindar 23
5.1.6 NC1 23
5.1.7 IP = PSPACE 23
5.1.9 MA (Merlin-Arthur) 24
5.2 Barriärer 24
5.2.1 Relativiseringsbarriären 24
5.2.2 Naturliga bevisbarriären 24
5.2.3 Algebrizationsbarriären 25
6 Konsekvenser av olika lösningar **27**
6.1 Konsekvenser om P ≠ NP bevisas 27
6.2 Konsekvenser om P = NP bevisas 27
6.3 Konsekvenser om varken P ≠ NP eller P = NP kan bevisas 27
7 Slutsats **29**
8 Referenser **30**

1 Introduktion

Cook och Levin var de första som formulerade $P \stackrel{?}{=} NP$ problemet på 1970-talet, men frågan har studerats längre än så. På 1950-talet hade både Gödel och Nash funderingar kring problemet innan det fått en konkret formulering. Problemet $P \stackrel{?}{=} NP$ frågar om det existerar en snabb algoritm för att hitta en lösning till ett matematiskt ja-nej-beslutsproblem med längden *n*.

År 1956 skrev Gödel ett brev till sin vän John von Neuman där han beskrev en stor oro vid tanken på en maskin med kapacitet att lösa matematiska problem. Gödel hade farhågor att det skulle innebära att matematiker inte längre skulle behöva tänka vidare på matematiska ja-och-nej frågor, om maskinen svarade nej på problemet skulle detta helt enkelt accepteras utan vidare granskning. Vissa har tagit detta ett steg till och är rädda att all kreativitet inom matematiken skulle dö ut. Den enda kreativiteten som eventuellt skulle kunna vara hotad är dock kreativitet rörande frågor vars svar snabbt kan verifieras av datorprogram [3, 12].

Det finns många anledningar till varför $P \stackrel{?}{=} NP$ är ett intressant problem. Om det skulle visa sig att algoritmen existerar samt är praktiskt användbar skulle det få konsekvenser för det datoriserade samhälle vi lever i idag. Det skulle inte längre spela någon roll om, exempelvis, e-post som skickas över internet krypteras. Vem som helst med tillgång till algoritmen skulle kunna läsa allt som skickades. Förutom detta är $P \stackrel{?}{=} NP$ ett av de sju Millennieproblemen vars lösning var för sig är värda en miljon dollar [1]. Landon Clay, en affärsman från Boston, är grundaren till Clay Mathematics institutet. Detta institut väljer ut Millennieproblemen och utlovar belöningarna. Om det skulle visa sig att P = NP kan, exempelvis, de resterande sex Millennieproblemen lösas med hjälp av den framtagna algoritmen. Ingen dålig kaffepeng!

Scott Aaronsons *$P \stackrel{?}{=} NP$* [1] och Richard J. Liptons *The P=NP Question and Gödel's Lost Letter* [15] används som standardreferenser.

2. Begrepp och definitioner kring $P \stackrel{?}{=} NP$

För att skapa förståelse kring P, NP och $P \stackrel{?}{=} NP$ ska vi gå igenom några relevanta begrepp och sammanhang. Oftast uttrycks $P \stackrel{?}{=} NP$ i termer av antingen språk eller beslutproblem, detta skall vi gå närmare in på innan vi går vidare in på komplexitetsklasser. Vi börjar med att behandla begreppen sträng och alfabet för att sedan kunna förklara vad som menas med ett språk i det här sammanhanget.

2.1 Strängar och alfabeten

En sträng tas fram genom att ta ett fixt antal element ur ett givet alfabet och sedan skriva dessa i en ordnad följd. Ett exempel på ett alfabet är det svenska alfabetet A till Ö, med ett alfabet kan ett oändligt antal strängar skrivas. Ett alfabet består alltså av en mängd symboler, symbolerna behöver inte bestå uteslutande av bokstäver utan kan också innehålla andra symboler såsom siffror.

Ett exempel på ett alfabet är alfabetet {0,1}, detta alfabet innehåller endast 0 och 1. Följande är ett exempel på en binär sträng, alltså en sträng som endast innehåller symbolerna 0 och 1, som är skriven i MATLAB:

```
exempel = "101001";
exempel()
```

När programmet körs i MATLAB kommer strängen “101001” att skrivas ut.

Språk 2.2

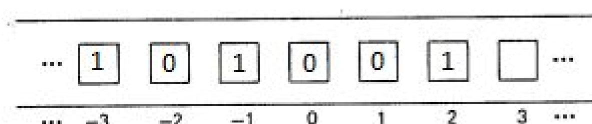
Datorernas grundspråk *S* består av endast av ettor och nollor, vilket var väldigt krångligt att skriva koder med. Modernare högnivåspråk såsom Python och C har tagits fram där både text och andra siffror kan skrivas in. Det finns också förprogrammerade kommandon att använda sig av för att tala om för programmet vad det skall göra. Detta har förenklat programmeringen avsevärt. Det finns också något som heter lågnivåspråk, det är ett språk som ligger nära hårdvarans (datorns) maskinkod. Dessa språk används för att skriva datorprogram. Ett datorprogram är ett antal instruktioner som beskriver vad datorn skall göra under tiden programmet körs, exempelvis addera 1 och 2. Skype och google chrome är två exempel på datorprogram. Högnivåspråk används för att bygga högnivåprogram och lågnivåspråk används för att bygga lågnivåprogram [10].

Högnivåprogram måste översättas till maskinkod, eller ett lågnivåprogram, för att programmet skall kunna köras. För vissa språk, som exempelvis C++, används en kompilator (ett datorprogram som översätter från ett språk till ett annat) för att översätta programmet till ett lågnivåprogram. För andra, såsom MATLAB, översätts språket under tiden programmet körs [10].

2.1 Turingmaskinen

P $\stackrel{?}{=}$ NP problemet förklaras ofta i termer av Turingmaskinen vilket är en teoretisk beräkningsmodell utvecklad av Alan Turing 1936. Förklaringen nedan är en sammanställning av Aaronsons och Wilfs [16] beskrivningar av turingmaskinen.

En turingmaskin består av en oändligt lång remsa som är uppdelad i diskreta rutor, varje ruta är numrerad. Varje ruta innehåller en symbol från ett alfabet som turingmaskinen känner till. För enkelhetens skull så kan vi säga att alfabetet består av “0”, “1” och symbolen för “tomrum”. Ett exempel på detta är följande figur:



Figur 1: En turingmaskinremsa.

Det finns också ett skriv- och läshuvud som kan både läsa och skriva en symbol i varje ruta, det kan också röra sig mellan rutor. Den kan endast flytta en ruta i taget åt antingen höger eller vänster.

En turingmaskin kan befinna sig i ett fixt antal tillstånd. r_0 står för ursprungstillståndet, en turingmaskin kan också förkasta eller acceptera en input. Ponera att vi vill veta om $x \in S$, för en binär sträng $x \in \{0,1\}^*$, där $\{0,1\}^*$ är mängden av *alla* strängar över alfabetet $\{0,1\}$ inklusive den tomma strängen. *x* kallas för en instans av det generella problemet som utgörs av att bestämma medlemskap i språket *L*. Låt *T(x)*, där *T* står för en turingmaskin, motsvara *Ts* körning med inputen *x*. *T(x)* kan antingen acceptera eller inte acceptera. Om *T(x)* accepterar innebär det att *T(x)* har stannat och hamnat i ett “acceptera”-tillstånd. Om, för alla $x \in \{0,1\}^*$

$$x \in S \Leftrightarrow T(x) \text{ accepterar}$$

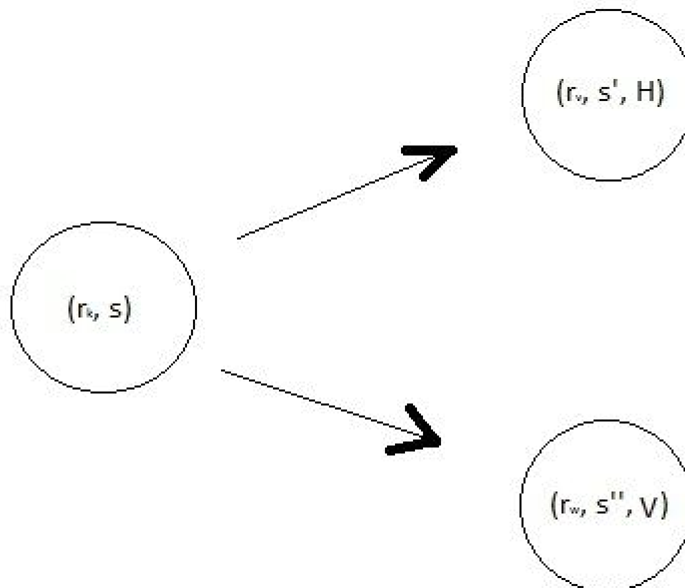
säger vi att M bestämmer språket *S*. *T(x)* accepterar *inte* om *T* stannar men hamnar i ett “förkasta”-tillstånd istället för ett “acceptera”-tillstånd. $r_1 ... r_n$ är de tillstånd som turingmaskinen hamnar i innan den antingen förkastar *x* eller accepterar *x*. Det är ett program som dirigerar maskinen genom varje steg i en uppgift, uppgiften kan exempelvis vara att ta reda på om $x \in S$.

Programmet använder sig av paret (*r*, *symbol*) för att bestämma vilket nytt tillstånd maskinen skall övergår till, vilken symbol maskinen skall skriva i den ruta befinner sig i samt om skriv- och läshuvudet skall flytta en ruta åt höger eller vänster.

Ett steg i programmet kan alltså skrivas som:

(*tillstånd*, *symbol*) → (*nytt tillstånd*, *ny symbol*, *steg*)

Vi har nu definierat en deterministisk turingmaskin. En icke-deterministisk turingmaskin kan övergå till flera olika tillstånd.



Figur 2: Exempel på två nya olika tillstånd för en icke-deterministisk turingmaskin.

Här kan alltså turingmaskinen gå från tillståndet r_k (k = 1,...,n) till antingen tillståndet r_v (v = 1,...,n) genom att skriva symbolen *s'* och ta ett steg till höger eller tillståndet r_w (w = 1,...,n) genom att skriva symbolen *s''* och flytta ett steg till vänster.

2.1.1 Church-Turing hypotesen

En välkänd hypotes, *Church-Turing hypotesen*, säger att alla beräkningsmodeller som är möjliga att definiera kommer att kunna simulera turingmaskinen och turingmaskinen kommer att kunna simulera modellen. Det finns numera ett modernt tillägg som går under namnet *Utvecklade Church-Turing hypotesen,* tillägget är att dessa simulationer kommer att dra ut som mest polynomiellt overhead i tid och minne. Det finns dock ett exempel på en beräkningsmodell inom kvantberäkning som överskrider tillägget i den *Utvecklade Chruch-Turing hypotesen.* Däremot, så länge vi talar om klassisk, digital och deterministisk beräkning så gäller den *Utvecklade Church-Turing hypotesen*.

2.2 P och NP definierade i språktermer

Nu kan vi definiera P och NP i språktermer. P utgörs av en klass med alla språk *S* för vilket det existerar en Turingmaskin *T* som bestämmer *S* i polynomielltid. Ovan är alltså *T* polynomielltid om det existerar ett polynom *p* så att *T(x)* stannar, accepterar eller förkastar, inom *p*(|x|) steg för alla x ∈ {0,1}* där |x| utgör *x*:s längd. Till exempel är |"00100"| = 5. NP är en klass med alla språk *S* för vilket det existerar en polynomielltid turingmaskin *T* och ett polynom *p* så att för alla *x* ∈ {0,1}*

$$x \in S \Leftrightarrow \exists v \in \{0,1\}^{p(|x|)} \text{ T } (x,v) \text{ accepterar},$$

där v är en polynomstor "vittnessträng". Alltså, om det för varje $x \in S$ existerar en "vittnessträng" v aktiveras "verifieraren" T av v och bekräftar om $x \in S$. Om svaret är ja så är S i NP. När $x \notin S$ får det inte finnas något v som gör att $T(x, v)$ accepterar.

Den önskade outputen är ja eller nej oavsett om P och NP uttrycks i språk eller beslutproblems termer. Om vi byter sikte från beslutproblem till sökproblem förändras inte P $\stackrel{?}{=}$ NP frågan alls, enligt följande sats.

Påstående 1 [Scott Aaronson] *Om P = NP, så finns det för alla språk $S \in NP$ (definierad av en verifierare T) en polynomielltid algoritm som hittar ett vittne, $v \in \{0,1\}^{p(n)}$, för alla $x \in S$, så att $T(x, v)$ accepterar.*

Bevis:
Idén går ut på att lära bitarna av ett accepterande vittne $v = v_1 \ldots v_{p(n)}$ en efter en, genom att fråga en serier av NP beluts frågor, till exempel:

- *Existerar det ett v så att $T(x, v)$ accepterar och $v_1 = 0$?*
  Om svaret är ja, fråga då:
- *Existerar det ett v så att $T(x, v)$ accepterar, $v_1 = 0$ och $v_2 = 0$?*
  Annars, nästa fråga:
- *Existerar det ett v så att $T(x, v)$ accepterar $v_1 = 1$ och $v_2 = 0$?*

Fortsätt på samma sätt tills alla p(n) bitarna av v har tagits fram.

Nu kan vi, istället för att endast få klart för oss om det finns en lösning på problemet eller ej, också ta reda på lösningen. För att ge en bättre förståelse av P $\stackrel{?}{=}$ NP problemet bör vi definiera ytterligare begrepp som har en nära relation till problemet.

2.3 Beslutsproblem

När vi talar om P $\stackrel{?}{=}$ NP i beslutsproblems termer säger vi att P, där P står för polynomielltid, är en klass av beslutsproblem. Dessa beslutsproblem är en oändlig mängd ja-och-nej frågor som kan lösas med hjälp av en digital dator (Turingmaskin) i polynomielltid. Termen polynomielltid innebär att maskinens beräkningstid ej kan överstiga inputens längd upphöjt till ett fixt tal. En input är data som matas in i ett program, körs och ger en output eller ett resultat. Ett program är en eller flera algoritmer som översatts till det programmeringsspråk eller beräkningsmodellen som användaren vill använda sig av.

Ett sätt att beskriva en algoritm på är att kalla dem instruktioner för ett antal relaterade beräkningsmetoder. Algoritmer står till grund för maskineriet i många företag idag. Till exempel kan Googles framgång till stor del tillskrivas deras sökalgoritm. Optimering är en också en viktig uppgift som algoritmer används till.

Ett exempel på en algoritm: Är talet jämnt eller udda?

Algoritm:

1. Läs in ett tal
2. Om talet är jämnt
   2.1. Skriv ut "Jämnt"
3. Om talet inte är jämnt
   3.1. Skriv ut "Udda"

Genom att skriva om algoritmen kan den användas i MATLAB:

Program i MATLAB:

```
prompt = 'Skriv ett tal:'
x = input(prompt);
if round (x/2) == x/2
        disp('Jämnt')
else
        disp('Udda')
end
```

Här är alltså inputen det tal vi vill testa om det är jämnt eller udda. Inputens längd beror på hur många siffror talet vi testar består av. Om x = 13 är längden 2, om x = 2 är längden 1 och så vidare. Outputen kommer att vara antingen "Jämnt" eller "Udda" beroende på inputen.

Nu kan vi skriva följande definitioner för P och NP:

Definition 1, P: *Klassen P består av alla beslutsproblem A för vilket det existerar ett polynom p och en turingmaskin som tar inputen x med längden n och bestämmer om x är i A eller ej, inom tidsbegränsningen p(n)* [3].

Definition 2, NP: *Klassen NP består av alla beräkningsproblem A för vilket det existerar ett polynom p och en icke-deterministisk turingmaskin som tar inputs med längden n och accepterar inputen x om och endast om $x \in A$, inom tidsbegränsningen p(n) [3].*

Definition 2 betyder alltså att om $x \in A$ så finns det en vittnessträng som verifierar att $x \in A$ på polynomiell tid.

2.4 Turingreduktion

Turingreduktion är en typ av polynomielltidreduktion, en polynomielltidreduktion går ut på att lösa ett problem genom att använda sig av en hypotetisk underrutin för att lösa ett annat problem. Underrutinen som görs är själva reduktionen och tiden som används är polynomiell.

Turingreduktion från ett problem P_1 till ett problem P_2 är en reduktion *f* så att *x* är i P_1 och om, endast om, *f(x)* är i P_2. Om en turingreduktion från P_1 till P_2 existerar kan varje instans för P_2 användas för att producera en instans för P_1. På samma sätt kan P_1 genomgå turingreduktion till P_3 om P_2 kan reduktioneras till P_3, med antagandet att en turingreduktion från P_1 till B är möjlig. P_1, P_2 och P_3 är då ekvivalenta, tre olika sätt att se på samma problem. På grund av detta kan de till synes helt olika beräknings problemen delas in i olika klasser. Det har också visat sig att naturliga beräkningsproblem kan delas in i ytterst få grupper, det finns alltså endast en handfull fundamentalt olika problem som är indelade i komplexitetsklasser [3].

2.5 NP-fullständiga och NP-svåra problem

Ett språk *L* är NP-fullständigt om alla problem i NP kan turingreduceras till *L*. Informellt så är NP-fullständiga problem de svåraste problemen i NP eftersom en effektiv algoritm för någon av dem skulle i sin tur ge effektiva algoritmer för alla NP problem. NP-svårt, också det informellt, betyder “minst lika svårt som något NP-problem under partiell ordning genom reduktion”. Det är inte fastställt att dessa indelningarna existerar, men under 1970-talet ledde forskningen fram till att flera hundra praktiska problem kunde delas in i dessa klasser.

Det som $P \stackrel{?}{=} NP$ frågar är alltså om det existerar en algoritm som för varje input storlek *n* löser ett NP-fullständigt problem i polynomielltid i *n*. Dock kan det vara mycket tidskrävande och onödigt komplicerat att använda samma algoritm för väldigt stora *n* som för små *n*. Aaronson tog fram ett exempel där han beskrev fördelen av att använda olika algoritmer beroende på storleken av *n*. Enklare, eller “naivare” algoritmer är effektiva när *n* är litet, men när *n* överstiger en viss storlek så kommer istället en “smartare” algoritm att vara effektivare. Genom att ha olika “smarta” algoritmer som tar över efter varandra beroende på vilken som är effektivast att, upp till väldigt smarta algoritmer, så minskas kravet på prestanda för de mindre storlekarna på n.

2.5.1 SAT och 3SAT

Ett exempel på ett NP-fullständigt problem är det så kallade SAT problemet. SAT står för booleskauppfyllningsproblem (the satisfiability problem).

Exempel 1:

(x ∨ y ∨ z) ∧ (**x** ∨ **y** ∨ **z**) ∧ (**x** ∨ y) ∧ (x ∨ **y**) ∧ (**y** ∨ z) ∧ (y ∨ z)

Exemplet ovan är ett SAT-problem, där varje parentes är klausul och x, y och z är booleska variabler. Symbolen ∧ är en konjunktion som representerar OCH och ∨ representerar ELLER. De booleska variablerna kommer att tilldelas sanningsvärden, alltså 1 för sant och 0 för falskt. **x** är x:s komplement, **y** är y:s komplement och **z** är z:s komplement . Om x tilldelas värdet 1 så kommer alltså **x** tilldelas värdet 0.

SAT [16]: Givet en mängd klausuler, existerar en tilldelning av sanningsvärden, en för varje variabel, så att alla klausuler får värdet 1?

SAT är uppenbarligen i NP. Ett vittne utgörs av just tilldelningen av värden på variablerna och vi kan snabbt kontrollera att alla klausuler blir sanna.

Exemplet ovan har ingen satisfierbar tilldelning. Minst en av x,y och z är sann och minst en är falsk för x = y och y = z.

Om vi exempelvis väljer tilldelningen x = 1, y = 1 och z = 0 får vi att:

$$(1 \vee 1 \vee 0) \wedge (0 \vee 0 \vee 1) \wedge (0 \vee 1) \wedge (1 \vee 0) \wedge (0 \vee 0) \wedge (1 \vee 0)$$

Här ser vi att klausulen ($\mathbf{y} \vee z$) blir (0 v 0) vilket gör att tilldelningen inte är satisfierad, värdet blir alltså falskt. Minst en av klausulerna kommer att vara falsk oavsett vilken tilldelnings som görs.

Detta problem är ett av de viktigaste (och svåraste) inom komplexitetsteorin. Kan vi hitta en algoritm som avgör SAT på polynomiell tid så har vi visat att P = NP.

3SAT består också av klausuler, precis som SAT, men varje klausul får som mest innehålla 3 stycken variabler eller komplement. SAT går att reducera till 3SAT, alltså får vi följande sats:

Sats 1 (Cook-Levin Satsen [6, 15]) 3SAT är NP-fullständigt.

Bevis [16]:

Låt en given instans av SAT innehålla en klausul

$$(x_1 \vee x_2 \vee \ldots \vee x_k), \quad \text{där } k \geq 4.$$

Klausulen kommer då att ersättas med k - 2 nya klausuler där variablerna y_i (i = 1,..., k - 3) används. De nya variablerna introducerar endast för detta syfte. De nya klausulerna är följande:

$$(x_1 \vee x_2 \vee y_1) \wedge (x_3 \vee \mathbf{y}_1 \vee y_2) \wedge (x_4 \vee \mathbf{y}_2 \vee y_3) \wedge \ldots \wedge (x_{k-1} \vee x_k \vee \mathbf{y}_{k-3})$$

Låt oss nu använda oss av följande antagande:

Antagande: Om $x_1^*,..,x_k^*$ är en tilldelning av sanningsvärden för x vilket gör att orginal klausulen (1) är sann, då existerar det argument $y_1^*,..., y_{k-3}^*$ av sanningsvärden till y så att alla k -2 nya klausuler (2) satisfieras av (x, y*) samtidigt.*

För att bevisa antagandet så måste vi först anta att (1) är satisfierad av en tilldelning x och att minst en av $x_1,...,x_k$ har värdet 1, vi döper en av dessa till x_r. Genom att tilldela y_z^* värdet 1 för $s \leq r - 2$ och y_s^* värdet 0 för s > r - 2 kan vi tillfredsställa (2).*

Om vi istället antar att alla $x_1,..., x_k$ är falska så säger antagandet att (2) är satisfierad att (2) vore satisfierad, med eller utan något av x:en. (2) kan då istället skrivas som:

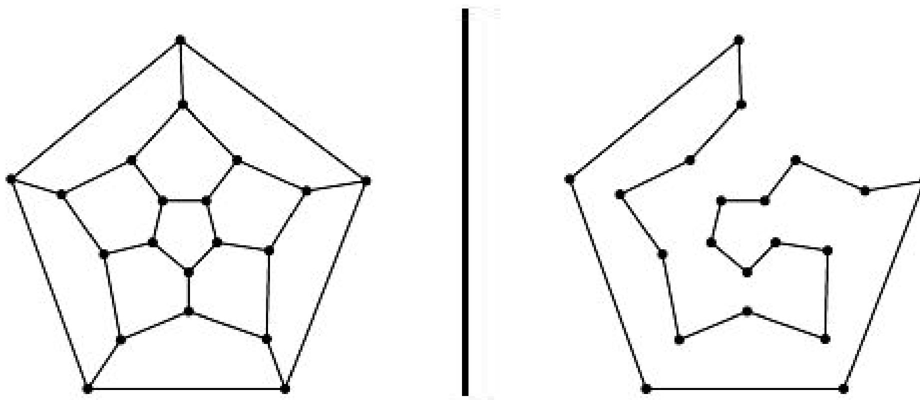
$$(y_1) \land (\mathbf{y}_1 \lor y_2) \land (\mathbf{y}_2 \lor y_3) \land \dots \land (\mathbf{y}_{k-4} \lor y_{k-3}) \land (\mathbf{y}_{k-3})$$

Om vi tittar på klausulerna ovan från vänster till höger kan vi se att y_1 måste vara sann, detsamma gäller för y_2 ända fram till y_{k-3}. Men y_{k-3} måste också vara falsk om ($\mathbf{y}_{k-4} \lor y_{k-3}$) skall vara satisfierad. Denna motsägelse visar att minst ett x:s måste vara sann. Denna transformation kan göras i polynomielltid, alltså kan SAT reduceras till 3SAT.

Enligt definitionen av NP-fullständighet så är alla problem som är NP-fullständiga även NP-svåra, SAT och 3SAT är alltså NP-svåra.

2.5.2 Hamiltoncykelproblemet

Hamiltoncykelproblemet frågar om en given graf *G* innehåller en väg som går genom alla hörn exakt en gång och dessutom slutar i samma hörn som vi började i [9].



Figur 3: Figuren visar ett Hamiltioncykelproblem.

Problemet uppenbarligen är i NP eftersom själva H-cykeln är vittne. Det visar sig att detta problem också är NP-fullständigt. Till exempel kan varje instans av SAT-problemet översättas till en graf *G* som har en H-cykel, om och endast om, SAT-instans har en satisfierad tilldelning av variabler.

2.6 coNP

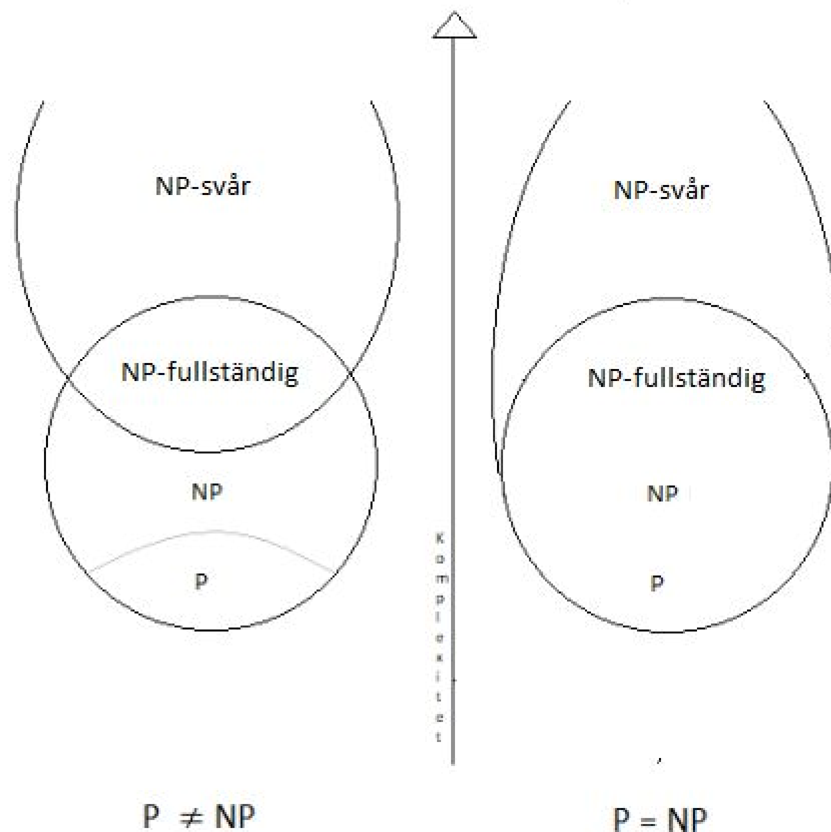
Låt $\mathbf{S} = \{0,1\}^* \setminus S$ vara komplementet av ett språk *S*, alltså de strängar som inte finns i *S*. coNP är en klass av alla språk $\mathbf{S}$ som är komplement till alla språk *S* som ingå i NP. Om $x \notin S$ och $\mathbf{S} \in$ coNP betyder det att det finns det ett kort samt snabbt kontrollerbart bevis för detta. Om P = NP så gäller P = coNP och NP = coNP. Om däremot P ≠ NP så gäller NP ≠ coNP. Notera att coNP inte är ekvivalent med **NP**, **NP** är en mängd med alla icke-NP problem.

Några exempel på coNP problem är komplementen till de problem och språk som tagits upp ovan. Komplementen till SAT och 3SAT är alltså coNP-fullständiga [11].

co-Hamiltoncykelproblemet skulle gå ut på att bevisa att en given graf *G inte* innehåller en hamiltoncykel. Vad skulle vittnet vara i det fallet?

2.7 NP-mellanliggande

Ett språk *S* är NP-mellanliggande om *S* varken är i P eller NP-fullständig men ingår i NP.



Figur 4: Figuren visar hur de olika grupperingarna är relaterade till varandra om P = NP eller om P ≠ NP.

Vi har då följande sats som också är illustrerad i figuren ovan:

Sats 2 ([Lander [13]) Om P ≠ NP så existerar det NP-mellanliggande språk.

Det betyder att om P = NP skulle det inte existera några NP-mellanliggande problem.

2.8 Polynomiell hierarki

Här introducerar vi begreppet turingmaskin med orakel. Vi tänker oss att oraklet utgörs av en svart låda som löser eller avgör en fråga på ett enda tidssteg. Till exempel blir nu A^L klassen av alla problem i *A* om vi har ett orakel för språket *L* och A^B är klassen av alla problem som ligger i A om vi har ett orakel för klassen av problem B.

Polynomiell hierarki (PH) är ett sätt att generalisera P, NP och coNP ytterligare. PH är en oändlig sekvens av klasser vars noll-nivå är ekvivalent med P och vars k:te nivå, där k ≥ 1, består av alla problem i P^S, NP^S eller $coNP^S$ för något språk *S* i nivån (k-1)^{rt}. Detta kan vi skriva som:

$$\Sigma_0^P = P, \quad \Sigma_k^P = P^{\Sigma_{k-1}^P}, \quad \Sigma_k^P = NP^{\Sigma_{k-1}^P}, \quad \Pi_k^P = coNP^{\Sigma_{k-1}^P},$$

Om P = NP är sant så är hela hierarkin ekvivalent, exempelvis:

$$\Sigma_2^P = NP^{NP} = NP^P = NP = P$$

Genom att visa att $\Sigma_k^P = \Pi_k^P$ eller att $\Sigma_k^P = \Sigma_{k+1}^P$ för något *k* kommer nivåerna att “kollapsa” till $\Sigma_k^P = \Pi_k^P$.

Antagande 1: *Alla nivåer hos PH är distinkta, alltså den oändliga hierarkin är strikt.*

Antagandet ovan är samma sak som P ≠ NP och många datavetare tror att detta är fallet, de anser att det vore konstigt om hierarkin kollapsade på en viss nivå men inte på nivåerna nedanför.

2.9 Polynomielltid

Hur kommer det sig att vi använder oss av begreppet polynomielltid? I avsnittet om turingreduktion togs det upp att turingreduktion är en typ av polynomielltidreduktion. Det är då inte endast polynom som har egenskapen att kunna genomföra reduktion med, men forskningen har lett fram till att det är polynomielltidreduktion som är den bästa funktionsgruppen för att karaktärisera naturliga problem [3].

Förutom fördelen ovan så är det också många program som har exekveringstiden (tiden det tar att köra programmet) n^2, där *n* är inputstorleken. Dessa vill vi inte exkludera ur klassen med “enkla” funktioner, men så fort kvadratisk tid tillåts så blir det väldigt svårt att inte tillåta polynomielltid. En funktion som beräknas i tiden n^2 kan ge en output i storleken n^2, att sedan beräkna f(f(x)) skulle ta tiden n^4. Detta leder alltså till att alla polynomielltid beräkningsfunktioner är enkla [3].

Genom att basera komplexitetsteorin på polynomielltid gör att vi kan nästa helt bortse ifrån vilken typ av dator som vi använder oss av när vi mäter exekeveringstiden. Samma sak gäller för valet av språk. Anledning är att nästintill alla “verklighetsbaserade” idéer som hittills föreslagits inom beräkning kan simuleras med endast en polynomiell bromsning av turingmaskiner. Detta är bakgrunden till *Church Turing hypotesen* [3].

3 $P \stackrel{?}{=} NP$:s betydelse

En lösning på $P \stackrel{?}{=} NP$ har stora möjligheter att påverka matematik, datorer och även resten av vårt datoriserade samhälle. Här tas även invändningar mot problemets relevans upp.

3.1 Betydelse för samhället

Vad skulle en lösning på $P \stackrel{?}{=} NP$ egentligen innebära för vårt samhälle? Om det visar sig att P = NP skulle det eventuellt ha konsekvenser på vårt samhälle och sätta att leva idag, men algoritmen måste isåfall vara praktiskt användbar och exekveringstiden måste vara rimlig. Mer om detta går att läsa i sektion 6.

3.1.1 Brute force sökningar och kryptering

Edmond var en av de första som började arbeta med problemet med brute force-sökningar, han skrev en polynomielltid algoritm för maximala matchningsproblemet. M är en matchning om M är en delmängd av den totala mängden av bågarna B i en graf $G = (N, B)$ där ingen av noderna N tillhör fler än en båge i M. En maximal matchning innebär att om det läggs till fler bågar i M slutar M att vara en matchning. Med bruteforce-sökningar menas att lösningen hittas genom att systematiskt söka igenom en hel mängd av tänkbara lösningar. Om det exempelvis är ett lösenord som eftersöks testas helt enkelt olika lösenord tills det rätta testas och hittas. Edmonds algoritm är ett strålande exempel på hur bruteforce-sökningar ibland kan undvikas. Om P = NP så skulle vi inte längre behöva utföra brute force sökningar på problem i P.

P = NP skulle innebära att de krypteringar som beror på beräkningsantaganden skulle knäckas, såsom kreditkortsnummer. Att hitta krypteringsnyckeln är ett NP-sökningsproblem. Notera att algoritmen måste praktiskt användbar för att detta skall ske. Som det nämndes i introduktionen skulle P = NP innebära att de andra sex Millennieproblemen kan lösas med hjälp av algoritmen, dessa skulle i sin tur påverka datavetenskapen [14].

3.1.2 $P \stackrel{?}{=} NP$ och A.I.

Problemet säger mycket om artificiell intelligens men inget om eventuella skillnader eller likheter mellan människor och maskiner. Om det skulle visa sig att P ≠ NP skulle det berätta mycket för oss om gränserna för all klassisk digital beräkning. Detta skulle kunna appliceras på både datorer och den mänskliga hjärnan. Men det enda som krävs av maskinerna för att konkurrera ut människan, ta över världen, är att vara intelligentare än människan. P = NP innebär alltså inte att maskinerna blir vår undergång.

3.2 Invändningar

Det finns åsikter om att problemet inte är så viktigt som det beskrivs och därför bör varken tid eller resurser läggas ned för att finna svaret. Nedan är de invändningar som enligt Scott

Aaronson vanligen dyker upp vid diskussioner kring $P \stackrel{?}{=} NP$ problemet samt respons till dessa.

3.2.1 Den asymptotiska-invändningen

$P \stackrel{?}{=} NP$ ger fortfarande inte information om hur många steg som faktiskt krävs i en uträkningen för konkreta värden på *n*, där *n* är längden på det matematiska påståendet. Detta anses relevant i praktiken. Det som $P \stackrel{?}{=} NP$ tittar på är om exekveringstiden av en algoritm växer polynomiellt eller exponentiellt när *n* går mot oändligheten, alltså tittar problemet endast på asymptoter.

Svar:
Antalet steg som behövs skiljer sig stort mellan dator-/maskinmodeller och andra faktorer som hur problemet är kodat, alltså det inte ett bra sätt att bedöma problemets svårighetsgrad. Asymptoternas komplexitet kan däremot ses som en rent matematiskt bidragande orsak till problemets svårighetsgrad. En asymptot är det värde som en funktion går mot då *n* går mot oändligheten. I praktiken finns det naturligtvis många andra bidragande orsaker till ett programs effektivitet men asymptoter har definitivt betydelse. Det finns också de som invänder mot att termerna *effektivitet* och *polynomiellt* anses nära relaterade samtidigt som *exponentiellt* och *ineffektiv* också anses relaterade i teoretisk datavetenskaps identifikation. Vore inte en algoritm som tar 1.00001^n steg, där *n* är ett godtyckligt tal, att föredra framför en som tar n^{1000} ? Dock är inte den typen av algoritmer så vanliga. Datorforskare har dessutom empiriska bevis för att det är en stark korrelation mellan "lösningsbar i polynomielltid" och "effektivt lösningsbar i praktiken".

3.2.2 Polynomiell-tid-invändningen

Varför skall gränsen för effektivitet dras vid polynomfunktioner och inte vid någon annan klass av funktioner - till exempel vid kvasipolynom funktioner av formen n^{log} ?

Svar:
Vid forskning kring algoritmer är målet att minska polynomets gradtal för problem som finns i P vilket gör att datavetare använder sig att exempelvis kvasipolynom när det behövs. Ett kvasipolynom har formen $n^{log^c n}$. Men i regel används alltså vanliga polynomfunktioner av följande anledningar: polynom är den minsta funktion klassen som försäkrar att en mängd av "effektivt lösningsbara" problem är oberoende av till exempel vilken maskinvara som används. En annan anledning är att klassen också innehåller linjära funktioner samt är slutna under grundläggande matematiska operationer såsom addition, multiplikation och sammansättning. Vid användandet av polynom kan alltså effektiva algoritmer sammansättas ett konstant antal gånger och sammansatta algoritmen är fortfarande, totalt sett, effektivt.

Det är värt att nämna att en stor del av forskningen kring algoritmer går ut på att sänka polynomets gradtal för problem som man vet ligger i P, här har forskarna fått mer utrymme när de gäller metoder för att sänka gradtalen. De kan alltså exempelvis använda sig av kvasipolynom när det behövs.

3.2.3 Diskbänksinvändningen

$P \stackrel{?}{=} NP$ ignorerar möjligheten att naturliga processer kan överskrida gränser för turingmaskiner som exempelvis analoga datorer. $P \stackrel{?}{=} NP$ är dessutom begränsad eftersom det endast berör diskreta samt deterministiska algoritmer. Diskreta algoritmer betyder att heltal används i algoritmerna.

Svar:
Nuförtiden finns det stora grenar inom den teoretisk dataforskning där det forskas om vad som händer om ett antagande bortses ifrån, såsom slumpmässiga algoritmer och approximerande algoritmer. Alltså det som togs upp ovan. Bland annat forskas det om probabilistiskt kontrollerbara bevis (PCP), det är en typ av bevis som kan kontrolleras av en randomiserad algoritm. En randomiserad algoritm är en algoritm som använder en viss del slumpmässighet, gärna likformigt slumpmässiga bitar, som en del av sin logik [8]. I det här fallet är slumpmässigheten begränsad och algoritmen läser endast en begränsad del av beviset. PCP och dess utlöpare har visat att för många NP-problem kan det inte ens existera en polynomielltid algoritm för att approximera svaret inom en rimlig faktor om inte P = NP; men exempelvis enligt P = BPP antagandet ger inte slumpmässiga algoritmer mer “kraft” än P samt noggranna analyser av brus, energiförbrukning med mera föreslår att det samma gäller för analoga datorer. BPP är en generalisering av klassen P där slump-metoder används och står för Bounded-Error Probabilistic Polynomial-Time. BPP är en klass av språk $S \subseteq \{0, 1\}^*$ för vilken det existerar en polynomielltid turingmaskin T och ett polynom p så att för alla inputs $x \in \{0, 1\}^n$ gäller:

$$\Pr_{r \in \{0,1\}^{p(n)}} [t(x,r) = S(x)] \geq \tfrac{2}{3}$$

Alltså, för varje x måste T bestämma (rätt) om $x \in S$ för de flesta val av r.

Enligt Aaronson är det många som påstår att även om $P \neq NP$ så kan vi nästan alltid hitta lösningar som är tillräckligt bra i praktiken, för de problem som är värda att bryr sig om. Genom att använda heuristisk är det till exempel möjligt att simulera genetik algoritmer. Vilket fungerar för vissa problem men inte alla. Hela syftet med kryptografi är att det inte skall fungera att lösa problem på detta sätt. Dessutom skulle det inte läggas tid på att lösa problem som anses “omöjliga” att lösa.

3.2.4 Den matematiska snobberi invändningen

Turingmaskinen är en mänsklig uppfinning vilket gör att $P \stackrel{?}{=} NP$ är inte ett “riktigt” matematiskt problem. Det handlar inte om ett naturligt matematiskt objekt som exempelvis heltal.

Svar:

$P \stackrel{?}{=} NP$ handlar inte alls om Turingmaskiner utan om algoritmer, Turingmaskiner är bara ett sätt att uttrycka dessa algoritmer. Invändningen i sig antyder att det fattas en viss allmän kunskap om hur långt forskningen kommit inom komplexitetsteorin. Mycket av komplexitetsteorin bygger på, för att nämna några exempel, Fourieranalys och algebraisk geometri. Detta är minst sagt matematik.

3.2.5 Surt-sa-räven-invändningen

$P \stackrel{?}{=} NP$ är inte värd vare sig ansträngning eller uppmärksamhet eftersom dess svårighetsgrad gör att de eventuella framsteg som kommer att göras ligger för långt in i framtiden.

Svar:
Även om $P \stackrel{?}{=} NP$ aldrig skulle lösas har problemet redan haft stor betydelse som inspiration när det gäller forskning inom algoritmer, kryptografi och många med teoretiska områden som datavetare arbetar med. Bara det gör att problemet är värt både ansträngning och uppmärksamhet. Dessutom vet vi mer nu om problemet än för några årtionden sedan.

3.2.6 Den självklara invändningen

Intuitionen säger att det är självklart att $P \neq NP$ och därför skulle lösningen knappt ge någon användbar information.

Svar:
Oftast när ett svar eftersöks inom matematiken ger det en hel del överraskningar på vägen, vilket redan uppenbarat sig i jakten på svaret för $P \stackrel{?}{=} NP$. För att kunna säga att $P \neq NP$ krävs kunskap om vad polynomielltid algoritmer *inte* kan göra men också en förståelse av vad de *kan* göra. Mer kunskap kring polynomielltid algoritmer kommer sannolikt att leda till fler algoritmer med praktisk betydelse. Dessutom är $P \stackrel{?}{=} NP$ långt ifrån det enda problemet inom matematik som denna typen av invändning skulle kunna appliceras på. Om denna tanke slog rot bland alla som forskade så vore Gödels farhågor redan uppfyllda.

3.2.7 Den konstruktiva invändningen

Även om $P = NP$ så finns det en risk att beviset för det blir icke-konstruktivt vilket skulle innebära vi inte skulle ha någon algoritm. Om så vore fallet skulle $P = NP$ inte ha någon väsentlig påverkan på samhället.

Svar:
Det finns absolut en teoretisk möjlighet att beviset ger en icke-konstruktiv algoritm, men detta skulle fortfarande innebära att det existerar en algoritm vilket skulle ge stor motivation till att hitta den nämnda algoritmen. På samma sätt skulle kunskapen om att beviset för $P = NP$ gav en n^{1000} algoritm men vi misstänker att en n^2 algoritm existerar. Då är det bara att fortsätta att forska.

4 Tro kring $P \stackrel{?}{=} NP$

Majoriteten av datavetare tror att $P \neq NP$. Den allmänna uppfattningen är att det existerar problem som går snabbt att kontrollera men tar lång tid att lösa. I de fallen är sökningar med bruteforce det bästa sättet att finna lösningen. Hur kommer det sig att, trots att det saknas bevis, att så många tror att $P \neq NP$?

4.1 Anledningar till att majoriteten av forskare tror att $P \neq NP$

När vi ställer oss frågan om $P \stackrel{?}{=} NP$ har lösningen $P = NP$ så frågar vi om, exempelvis, SAT-lösare *alltid* vid polynomielltid fungerar och om det *alltid* finns smarta sätt att undvika exponentiella sökningar. Vid tron att det existerar en kryptografisk envägsfunktion såsom $x \rightarrow f(x)$ som är enkel att beräkna men svår att invertera räcker det för att få uppfattningen att $P \neq NP$. Det finns två väldigt starka argument för att $P \neq NP$. Det första är att om $P = NP$, varför har då ingen i hela mjukvaruindustrin kommit fram till något som skulle kunna användas till att ta fram en algoritm för att invertera godtyckliga envägs funktioner? I många fall i verkligheten såsom i linjärprogrammering (bivillkor och målfunktionen är linjära funktioner) utvecklades snabba metoder för att hantera sådana problem årtionden innan någon till fullo förstod teorin bakom. Hur kommer det sig att inga sådana metoder uppenbarat sig i den här frågan? Det andra argumentet går ut på att tusentals problem har visats vara NP-fullständiga och tusentals har visat vara i P, men inte ett enda problem har kunnats visas vara i både P och NP. Det hade räckt med ett problem för att teorin om att $P = NP$ skulle stärkas. Det här fenomenet blir extra slående när vi tittar på approximationsalgoritmer för NP-hårda problem som återger en lösning som iallafall har en del optimalitet. Se satsen nedan.

Sats 3 (Håstad [12]) *Anta att det finns en polynomielltid algoritm som, given en satisfierbar 3-SAT instans φ som input, ger en output i form av en tilldelning som satisfierar minst $7/8 + \varepsilon$ av klausulerna, där konstanten $\varepsilon > 0$. Då är $P = NP$.*

4.2 Konsekvenser av förutfattade meningar

Eftersom en majoritet tror att $P \neq NP$ kan det medföra konsekvenser, tänk om de tror fel? Nedan finns en lista på tre konsekvenser som bias kan leda till som Lipton tagit fram, listan utgår ifrån att majoriteten tror att ett obevisat problem *X* är sant:

(1) Få arbetar med att bevisa att *X* är falskt, vilket kan leda till att det slutgiltiga resultatet dröjer.
(2) Fältet kanske baserar resonemang på *X*, exempelvis "Om X är sant, är Y sant", dessa resonemang är i grunden felaktigt.
(3) Även om *X* är sann, genom att tänka på *X* men också dess negation kunde fältet ha upptäckt andra viktiga bitar som annars skulle ha missats.

4.2.1 LBA problemet

LBA problemet är ett exempel på ett problem där övervägande delen forskare var övertygade om vilken lösning problemet skulle ha, men hade fel. Nedan följer en redogörelse.

LBA står för Linear Bounded Acceptor och fungerar som en icke-deterministisk Turingmaskin vars remsa är exakt samma längd som inputen. Om ett språk *S* accepteras av en LBA så är *S* kontextberoende. LBA problemet frågar alltså om en sådan maskin är sluten under komplement, med andra ord om *S* accepteras, accepteras då även komplementet till *S*?

S.Y. Kuroda var den som först tog upp problemet 1964, på den tiden fanns det ett starkt intresse kring grammatik som används för att beskriva språk. En av de första frågorna som det letades svar på var; Om *S* är ett kontextberoende språk, är dess komplement ***S*** också det? Majoriteten av datavetarna var övertygade om att svaret var nej.

LBA problemet verkade så svårt att ingen jobbade med det och man var övertygad om att resultatet skulle gå åt ett visst håll, så vad var vitsen att lägga ner energi på att bevisa det? Robert Szelepsczényi var en student när han löste problemet, det ryktas att han fått ett gäng uppgifter att lösa men eftersom han inte varit på föreläsningen visste han inte att den sista var det kända LBA problemet utan satte sig ner och försökte lösa uppgiften, och lyckades.

Majoriteten av datavetarna var alltså övertygade om att svaret var nej på LBA problemet och till allas förvåning var svaret ja. Tänk om samma sak gäller för $P \stackrel{?}{=} NP$!

5 Barriärerna

Komplexitet teoretiker har identifierat tre tekniska barriärer: relativisering, naturliga bevis och algebrasering som ett bevis för P ≠ NP måste bemöta. Det har visats att det är möjligt att bemöta barriärerna, men inte alla tre samtidigt. Enligt Aaronson är anledningen till att P ≠ NP är så svårt att bevisa att i fall efter fall så finns det väldigt smarta sätt att undvika bruteforce-sökningar, mångfalden av de sätten rivaliserar med mångfalden inom matematiken själv. Och även om det verkar finnas ett osynligt staket som separerar NP-fullständiga problem från den lilla variationen mellan dem i P så skulle alla argument som vi kan föreställa oss för varför NP-fullständiga problem är svåra också kunna appliceras på varianterna i P.

5.1 Ytterligare begrepp

Ytterligare begrepp bör definieras för att ge en förståelse kring de tre olika barriärerna.

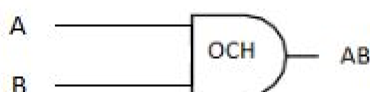
5.1.2 Orakelturingmaskin

Vi nämnde orakel i kapitel 2.8, låt oss kort repetera begreppet inför kommande diskussion. För att kunna förklara begreppen måste vi först gå igenom vad en *orakelturingmaskin* är för något. Om en turingmaskin kan avge en instans *x* till en apparat som inom ett enda tidssteg återger ett svar som indikerar om $x \in S$, så är apparaten för ett "orakel". Det är orakelturingmaskiner som möjliggör att vi kan relatera olika beräkningsproblem till varandra. S-orakel kallas de orakel som svarar på alla förfrågningar med *S*, S-orakel-Turingmaskiner noteras som T^S. Klassen av alla språk *S'* för vilket det existerar en T^S som bestämmer *S'* i polynomielltid definierar P^S.

Vi kan tänka oss ett orakel som ett oändlig samling av booleska funktioner $f_n : \{0,1\}^n \rightarrow \{0,1\}$ för varje n.

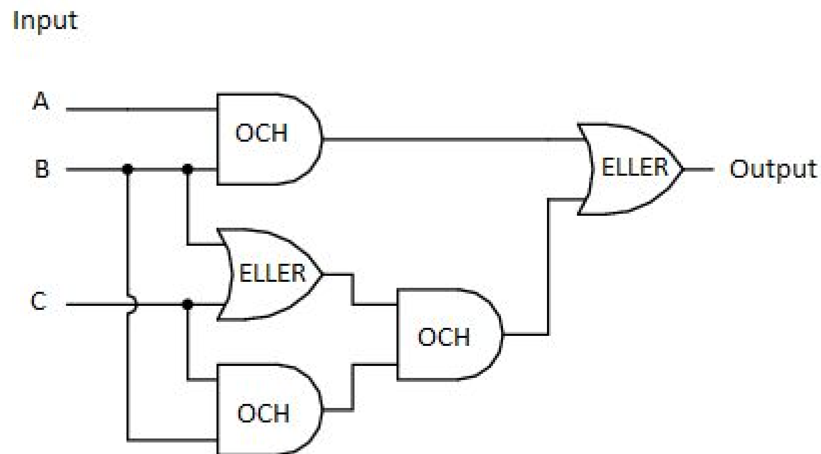
5.1.3 Booleska grindar

Exempel på booleska logik grindar är *OCH*, *ELLER* och *INTE*. Nedan visas en boolesk OCH grind som visar sanningstabellen ovan i form av en grind.



En krets består av dessa grindar [5]. En krets storlek är ekvivalent med antalet grindar som bygger upp kretsen. Maxantalet av input strängar som kan gå igenom en grind g kallas för kretsens fanin. Fanout är maxantalet strängar som kan komma ut ur en grind *g*. Fanout är

också samma sak som antalet grindar som är i direkt beroende av grindens *g*s output. Nedan syns en karta över ett grindnät med OCH samt ELLER grindar.



5.1.3 P/poly

P/poly är en klass med språken *S* och för vilket det existerar ett en polynomielltid turingmaskin *T* samt en oändlig mängd med *n* stycken “rådsträngar” a_1, a_2,... där a_n är *p(n)* bitar lång för något polynom *p* så att för alla *n* och alla $x \in \{0,1\}^n$, har vi

$$T(x,a_n) \text{ accepterar } \Leftrightarrow x \in S.$$

P/poly kan också definieras som en klass av språk *S* som känns igen av en familj med polynomiellt stora booleskt grindnät. Det finns en grindnät för varje *n*. P/poly kallas för den icke-likformiga generaliseringen av P. Ordet “icke-likformig” kommer från att det finns en grindnät som inte nödvändigtvist måste ha samma struktur för varje *n*.

5.1.4 AC^0

Låt AC^0 vara en klass med språk $S \subseteq \{0,1\}^*$ för vilket det existerar en familj med grindnät $\{G_n\}_{n\geq 1}$, en för varje input storlek *n*, så att:

(1) $G_n(x)$ ger outputen 1 om $x \in S$ och 0 om $x \notin S$, för alla *n* och $x \in \{0,1\}^n$.

(2) Varje G_n består av obegränsade-fanin AND och OR portar, såväl som NOT grindar.

(3) Det existerar ett polynom *p* så att varje G_n har som mest *p(n)* stycken grindar.

(4) Det existerar en konstant *d* så att varje G_n har högst djupet *d*.

Genom att bortse från det fjärde villkoret så har vi alla villkor från P/poly, AC^0 är alltså en underklass till P/poly. Det forskades mycket kring AC^0 på 1980-talet och det finns en god uppfattning om vilka problem som ingår i AC^0 och inte. Dessutom vet vi att $NP \not\subset AC^0$. P/poly har varit desto svårare att förstå sig på.

5.1.5 TC^0-kretsar och MAJORITET grindar

TC^0-kretsar är polynomiell stora med konstant djup med obegränsad fanin med MAJORITET grindar. MAJORITET grindar ger outputen sant om mer än hälften av inputen är sann och falskt om mindre än hälften av inputen är sann.

5.1.6 NC^1

NC^1 är en klass med problem som går att lösa med ett polynomiellt antal parallella processorer i polylogaritmisktid. NC^1 är en underklass till P/poly och och i termer av språk kan man säga att klassen endast innehåller språk *S* som är bestämda av en familj av kretsar i polynomiell storlek med djupet *O(log n)*.

5.1.7 IP = PSPACE

På 1980-talet väcktes ett intresse för interaktiva bevissystem. Interaktiva bevissystem är ett protokoll där en icke-trovärdig verifierare försöker övertyga en skeptisk polynomielltid verifierare om att ett matematiskt påstående är sant. Den förstnämnda verifieraren kallas Merlin och den skeptiska kallas Arthur. Arthur genererar slumpmässigt utmaningar och utvärderar sedan Merlins svar på dessa. Verifieringen sker alltså som i en dialog mellan Arthur och Merlin.

Låt IP vara en klass av språk $S \subseteq \{0,1\}^*$ för vilket det existerar en probabilistisk polynomielltids algoritm som Arthur har. Arthur får en input sträng $x \in \{0,1\}^n$ och genererar sedan upp till $n^{O(1)}$ utmaningar till Merlin. Varje utmaning består av en sträng med upp till $n^{O(1)}$ bitar. Merlin är medveten om Arthurs input sträng. Merlin svarar på utmaningarna och försöker övertyga Arthur om att $x \in S$. Det är sedan upp till Arthur att acceptera eller förkasta Merlins påstående.

PSPACE är en klass med språk S som kan bestämmas av en turingmaskin som använder sig av ett polynomiellt antal bitar av minnesutrymme. Det finns inga restriktioner för antaget tidssteg. Eftersom en icke-parallell algoritm kan få tillgång till maximalt n stycken minnesceller under n tidssteg vet vi att $P \subseteq PSPACE$ och på så vis att $P \subseteq NP \subseteq PH \subseteq PSPACE$.

Om vi endast tillåter Merlin att skicka ett meddelande till Arthur som Arthur sedan får ta sitt beslut från så har vi istället NP, IP är alltså en generalisering av NP. Det gör att $IP \subseteq NP \subseteq PSPACE$.

5.1.9 MA (Merlin-Arthur)

Låt MA vara en klass med språk $S \subseteq \{0,1\}^*$ för vilket det existerar en probabilistisk polynomielltidsverifierare *T* och och ett polynom *p* så att det för alla inputs $x \in \{0,1\}^*$ gäller:

1. Om $x \in S$ existerar det en "vittnessträng" $v \in \{0,1\}^{p(|x|)}$ så att sannolikheten att T(x, v) accepterar är minst ⅔.

2. Om $x \notin S$ så är sannolikheten att T(x,v) förkastar minst ⅔ för alla v.

MA innehåller både NP och BPP.

5.2 Barriärer

Här följer en redogörelse för de tre barriärerna som en lösning på P ≠ NP måste överkomma.

5.2.1 Relativiseringsbarriären

Baker, Gill och Solovay kom fram till relativiseringsbarriären 1975. Låt *C* och *D* vara två komplexitetsklasser. De tekniker som används för att bevisa att exempelvis $C \subseteq D$ eller $C \not\subseteq D$ är väldigt generella och fungerar ibland inte alls. Det gör att teknikerna kan bevisa att $C^L \subseteq D^L$ eller $C^L \not\subseteq D^L$ för alla möjliga orakel *L*. Alltså, om alla maskiner som finns med i beviset fick tillgång till samma orakel är beviset helt omedvetet om den förändringen och gås igenom på samma sätt som innan. Bevis som "utsätts" för detta sägs "relativisera", eller med andra ord vara "relativt för något orakel". En vidare förklaring är att tänka sig att bevisen använder en turingmaskin T_1 för att simulera en annan turingmaskin T_2. Om T_2 får tillgång till oraklet *L* så kan T_1 fortfarande simulera T_2 såvida T_1 också får tillgång till *L* som används att simulera T_2:s orakels anrop. Innan detta sker har T_1 inte alls tittat på T_2:s inre struktur [3].

Tillvägagångssättet har vissa begränsningar eftersom det också kan bevisa andra, starkare påståenden, som inte fanns med i beräkningen. Om det starkare påståendet är sant, men det svagare är falskt så kan resultatet bli missvisande och samma sak gäller för om det svagare påståendet är falskt och det starkare är falskt.

Det visade sig att $P \stackrel{?}{=} NP$ erkänner "motsägelsefulla relativiseringar": i några orakels världar gäller P = NP men i vissa är P ≠ NP vilket medför att teknikerna som fungerar för vissa orakel inte fungerar för andra. Detta gjorde att Baker, Gill och Solovay kom fram till att ingen relativiseringteknik kan användas för att lösa $P \stackrel{?}{=} NP$ problemet.

Sats (Baker-Gill-Solovay [4])

Det existerar ett orakel L så att $P^L = NP^L$, och ett annat orakel M så att $P^M \neq NP^M$.

5.2.2 Naturliga bevisbarriären

Razaborov och Rudich döpte barriären till naturliga bevisbarriären år 1993, men Michael Sipser kände till barriären redan på 1980-talet [16]. Det hela började med att man började

titta på AC^0- och $AC^0[p]$-kretsars undre gräns. Det upptäcktes att den undre gränsen för starkare och starkare klasser av kretsar kunde hittas genom att mer och mer generalisera den slumpmässiga restriktionen och polynommetoderna.

Slumpmässig restriktion är en kombinatorisk teknik, den typen av teknik kan ibland ställa till det för sig själv. De visar dels att en specifik funktion är svår för AC^0, men också om en slumpad funktion är svår för AC^0. Teknikerna skapar en algoritm vars input är sanningstabellen för en boolesk funktion $f : \{0, 1\}^n \rightarrow \{0, 1\}$. Om beviset ger upphov till en algoritm som uppfyller följande kriterier;

(1) **Konstruktivitet**: Algoritmen körs i polynomielltid och har samma storlek som f:s sanningstabell, alltså 2^n.
(2) **Storhet**: Om f väljs likformigt rent slumpmässigt, då är sannolikheten att algoritmen säkerställer att f är svår minst $1/n^{O(1)}f$.

så är det beviset för undre gränsen naturligt, enligt Razborov och Rudich.

Steg ett vore att generalisera polynommetoden som hanterar $AC^0[m]$ kretsar, där *m inte* är produkten av ett naturligt heltal upphöjt till ett fixt tal. (Ett exempel på ett sådant tal är 9 eftersom $9 = 3^2$.) Steg två vore att hantera TC^0 kretsar. Därefter är det NC^1 klassen som gäller. Till sist är det dags för klassen P/poly. Detta verkade vara en möjlig metod för att bevisa att $NP \not\subset P/poly$, alltså att $P \neq NP$, men redan vid TC^1 stöter metoden på naturliga bevisbarriären, om inte tidigare.

Men som vi nämnde ovan kan denna metod ställa till det för sig själv. Problemet är att en pseudoslumpmässig funktion, där $f : \{0, 1\}^n \rightarrow \{0, 1\}$, går inte att skilja från “verkligt” slumpmässig funktion. Antag att vi har ett naturligt nedåtbegränsningsbevis mot kretsklassen *K*. Definitionen säger då att vi har en effektiv algoritm *A* som försäkrar att $f \notin K$ med sannolikheten $1/n^{O(1)}f$. Innebörden av detta är att K inte kan innehålla väldigt starka familjer av pseudoslumpmässiga funktioner. *A* kan, till skillnad från om *f* är verkligt slumpmässig, aldrig säkerställa att $f \notin K$ om *f* är en pseudoslumpmässig funktion som är definierad i *K*. Eftersom verkligt slumpmässigaa och pseudoslumpmässiga funktioner inte går att skilja åt fungerar alltså inte den här typen av bevis för att avgöra $P \stackrel{?}{=} NP$.

5.2.3 Algebrizationsbarriären

Avi Wigderson och Scott Aaronsson kom fram till algebrization barriären 2008. Algebrization barriären är en modifikation av relativiseringsbarriären och namnet är en sammandragning av algebraic och relativization [2].

De visade sig att för att kunna bevisa $P \neq NP$ behövs tekniker för att undvika, förutom de föregående barriärerna, algebrizationsbarriären. För att kunna förklara vad som menas med algebrizationsbarriären måste vi först förklara några begrepp. Först och främst så har booleska funktioner f: $\{0,1\}^n \rightarrow \{0,1\}$ låggradsutvidgningar $\mathcal{F}(x) : \mathbb{F}^n \rightarrow \mathbb{F}$, av *f* över någon

stor ändlig kropp $\mathbb{F}$. Ett vanligt orakel består av en oändlig familj av booleska funktioner f_n och alla dessa har låggradsutvidgningar $\tilde{F}_n$. Ett algebraiskt orakel ger oss tillgång till både f_n och $\tilde{F}_n$. Utvidgningen måste vara ett polynom av låg grad, max 2n och måste ha formen:

$$\tilde{F}_n(x) : f_n(x) \ \forall \ x \in \{0,1\}^n$$

Sådana förlängningar existerar alltid.

Om vi vill utvärdera både boolesk och icke-boolesk input, så som IP = PSPACE, och vi har en formel eller krets för f_n kan vi använda algebraiska orakel. Som exempel, givet en 3-SAT formel ρ så kan vi "lyfta" ρ till ett låg grads polynom ϱ över en ändlig kropp $\mathbb{F}$ genom att omtolka AND,OR och NOT grindar i termer av kropp addition och multiplikation. Så om vi försöker att fånga styrkan av beräkning relativt till en orakel funktion f_n, då borde vi också kunna lyfta f_n. När vi lyfter f_n så får vi att icke-relativiserings resultat som är baserade på Gödeltal, såsom IP = PSPACE, relativiserar med hänsyn till algebraiska orakel:

Sats *P^{A} = $PSPACE^{A}$ för alla algebraiska orakel A. Liksom $PSPACE^{A} \subset P^{A}$ /poly implicerar att $PSPACE^{A}$ = MA^{A} för alla algebraiska orakel A, och så vidare, för alla interaktiva bevis resultat.*

Om vi vill formulerar om en formula φ, som innehåller A-orakel grindar i ett interaktiv protokoll, till gödeltal kan vi alltså hantera icke-booleska inputs till A-oraklet genom att kalla på A.

Sats (Aaronson-Widgerson[2]): *Det existerar ett algebraiskt orakel A så att P^{A} = NP^{A}. En konsekvens av detta är att ett bevis för P ≠ NP kräver icke-algebrization tekniker.*

6 Konsekvenser av olika lösningar

Det finns tre möjliga lösningar på $P \stackrel{?}{=} NP$ problemet, dessa tas upp nedan samt deras eventuella konsekvenser.

6.1 Konsekvenser om P ≠ NP bevisas

Konsekvenserna om P ≠ NP är i princip inga. Kryptering skulle kunna fortsätta som vanligt, det som är nämnvärt är återigen att vägen hit kan ge andra insikter inom området samt en större förståelse kring eventuella begränsningar inom A.I.

Det är viktigt att ha i åtanke att om man upptäcker effektiva (polynomiella) algoritmer för SAT så måste algoritmen fungera för alla instanser. För att P = NP måste algoritmen alltid fungera, utan undantag! Det skulle räcka med att algoritmen ger ett enda fel svar för någon instans för att P ≠ NP. Det finns såklart en risk för att detta inte märks, det kan vara ytterst sällan det sker. Det är dock inte samma sak som att P = NP, för att P = NP måste alla problem i PH ha en polynomielltids algoritm [3].

6.2 Konsekvenser om P = NP bevisas

Ett bevis för att P = NP behöver inte betyda att det i sig ger så stor påverkan på samhället, om någon alls. Även om vi lyckas ta fram en algoritm kanske dess exekveringstid är enorm vilket skulle göra den opraktisk att använda. Eller så kanske vi endast finner bevis för att en effektiv algoritm existerar men beviset i sig saknar ledtrådar som kan leda fram till algoritmen. Men även om så vore fallet så finns det alltid hopp. Till exempel är det väldigt ovanligt att så otympliga algoritmer tas fram för naturliga problem utan att en effektiv algoritm hittas. Dessutom, om det finns bevis för att en algoritm existerar så kommer motivationen för att hitta algoritmen med stor sannolikhet att öka vilket gör att chansen att faktiskt finna algoritmen också ökar [3, 12].

Men självklart kan det också vara så att beviset innehåller en effektiv algoritm, exempelvis linjär eller kvadratisk. Det skulle leda till att all matematik kan automatisera samt att många sysslor i A.I. skulle trivialiseras, såsom inlärning. Optimering vore en ren rutin medan kryptografi vore omöjligt med dagens metoder [3].

6.3 Konsekvenser om varken P ≠ NP eller P = NP kan bevisas

Tänk om det existerar ett snabbt program för SAT, men det finns inget sätt att bevisa att det verkligen fungerar. Ett sådant program skulle kunna användas för att besluta om en SAT formel är satisfierbar eller ej och sedan att ta fram tilldelningar som var satisfierande för SAT om formeln är satifieringsbar. Men hur skulle vi kunna veta att programmet har rätt om det beslutar att formeln inte kan satisfieras [3]?

Det vanligaste tillvägagångssättet för att bevisa att ett påstående ф är oberoende av något formellt system *L* går nästan alltid de facto ut på att bevisa att ф är oberoende av ett starkare formellt system som tas fram genom att genom att förlänga *L*s med alla sanna första ordningens logiska påståenden som innehåller endast universella kvantifierare. Detta spelar roll eftersom om det är känt att P ≠ NP är oberoende av detta starkare formella system kan vi bevisa att SAT måste ha booleskakretsar som är nästan polynomiellt stora. Alltså, de har en kretsstorlek av storleken $n^{\omega(n)}$ där ω är en mycket långsamt växande funktion d.v.s. ej begränsat av något polynom. Det betyder att om P ≠ NP är oberoende av starkare system så är det nästan samma sak som att SAT är enkelt att avgöra (om det är satisfierbart eller ej) ändå. Ett bevis för att varken P ≠ NP eller P = NP går att bevisa kommer med stor sannolikhet att dröja [3].

7 Slutsats

Det har forskats mycket kring P $\stackrel{?}{=}$ NP frågan och än är det långt kvar. Arbetet har än så länge tagit oss fram till tre, än så länge, svår överkomliga barriärer. Relativisering-, naturliga bevis samt algebrizationsbarriären. Bara detta har gett oss ytterligare kunskap om komplexitetsteorin och vi har fått information om varför de tillvägagångssätt som används stött på dessa hinder.

Även om en majoritet av datavetarna har uppfattningen att vi kommer att komma fram till resultatet P ≠ NP så finns det ännu inga bevis för detta. Till mångas förfäran, i alla fall de som arbetar med kryptografi, kan lösningen vara P = NP! Detta skulle kunna underlätta i arbetet med A.I. och inom industrin, men också stjälpa informationshanteringen på internet. Men för att detta skall ske så måste alltså beviset ge en algoritm så både är effektiv och går att kontrollera så att den fungerar för alla instanser.

8 Referenser

[1] Scott Aaronson. *$P \stackrel{?}{=} NP$*, sida 3 - 99. University of Texas at Austin, 2017.

[2] S. Aaronson and A. Wigderson. *Algebrization: a new barrier in complexity theory.* ACM Trans. on Computation Theory, 2009.

[3] Eric Allender. *Advances in Computers,* sida 117-147. Rutgers University, 2009.

[4] T. Baker, J. Gil och R. Solovay. *Relativizations of the $P \stackrel{?}{=} NP$ question*, volym 4 sida 431 - 442. SIAM J. Comput., 1975.

[5] Jehoshua Bruck (a), Marc D. Riedel (b). *Discrete Applied Mathematics*, sida 1877-1900
(a) Electrical Engineering, California Institute of Technology, Pasadena, CA 91125, USA, 2012.
(b) Electrical and Computer Engineering, University of Minnesota, Minneapolis, MN 55455, USA, 2012.

[6] S. A. Cook. *The complexity of theorem-proving procedures* sida 151-158. In Proc. ACM STOC, 1971.

[7] S. Cook. *The P versus NP problem*, Clay Math Institute official problem description, 2000.

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest och Clifford Stein. *Introduction to Algorithms*, andra utgåvan, kapitel 5: *Probabilistic Analysis and Randomized Algorithms*, sida 91-122. MIT Press and McGraw–Hill, 1990.

[9] Tomáš Dvořák, Jiří Fink, Petr Gregor, Václav Koubek. *Discrete Mathematics,* sida 2599-2611. Charles University, 2011.

[10] David K. Gifford, Mark A. Sheldon och Franklyn A. Turbak. *Design concepts in programming lanuguages*. Cambridge, 2008.

[11] Harald Hempel , Michael Krüger. *Theoretical Computer Science,* sida 49. Friedrich-Schiller-Universität Jena, 2008.

[12] J. Håstad. *Some optimal inapproximability results*, sida 48:798-859. J. of the ACM, 2001.

[t13] R. E. Ladner. *On the structure of polynomial time reducibility, volym 22* sida 155-171. J. of the ACM, 1975.

[14 L. A. Levin. *Universal sequential search problems, Problems of Information Transmission,* volym 9, sida 115-116. , 1973.

[15] Richard J. Lipton. *The P=NP Question and Gödel's Lost Letter,* sida 9 - 83. New York: Springer 2010.

[16] A. A. Razborov and S. Rudich. *Natural proofs. J. Comput. Sys. Sci.,* 55(1):24–35, 1997.

[17] Hybert S. Wilf. *Algorithms and Complaxity.* University of Pennsylvania, Philadelfi, 1994.