P V NP: EN FÖR ALLA?

A PREPRINT

Student:
Mbwenga Maliti
NANAT16
Huddinge Gymnasium
Huddinge, Gymnasievägen 3
mbwenga.maliti@edu.huddinge.se

Handleda re:
Karl Stengård
NANAT16
Huddinge Gymnasium
Huddinge, Gymnasievägen 3
karl.stengard@edu.huddinge.se

April 22, 2019

ABSTRACT

The purpose of this essay is to examine how the computational complexity inherent to algorithms used for different problems in computer science affects their running time, and in some cases accuracy, along with it's connections to technological challenges currently being tackled in today's society. We conduct a brief survey of key concepts related to the design of software algorithms and data structures, test the performance of well-studied sorting and regression algorithms, and encourage further reading via citations of books, journals and more. I chose this topic for my essay due to my interest in mathematics, which eventually lead me into studying software engineering, computer science and artificial intelligence on my free time, allowing me to now in addition to my school studies also work part-time as a software developer at DigitalRoute AB since October 2018.

1 Inledning

Under en vanlig dag så äter dom flesta frukost, kanske kollar nyheter på mobilen, tar bussen till skolan etc. Under varje steg tillfredställer vi något behov, vare sig om det är transport, nyfikenhet eller hunger. Processen att ta sig från behov till tillfredsställelse kan därför mer eller mindre anses vara lösningen till ett problem. Beroende på problemets komplexitet, kan mängden steg som behövs för att utföra den optimala lösningen variera kraftigt. Varje steg kan i vissa fall i sin tur sönderdelas i flera nya steg, vilket formger en uppsättning nya problem under ytan av det första. Exempelvis så anser dom flesta att konsumtionen av mat är en simpel process, där man helt enkelt tar en sked, gaffel eller liknande, lastar den med en lämplig mängd, för den in i sin mun, och sväljer. Dock pågår parallellt en hel skara av intrikata biologiska processer, såsom metabolism, som ser till att rörelsen, tanken att utföra den, och utnyttjandet av resurserna inuti maten kan ta plats. Lyckligtvis behöver vi ej medvetet smälta den mat vi äter, dock kan detsamma inte sägas för budgeterandet av månadsinkomster eller renhållning av utrymmen. En möjlig lösning till strävandet för en tankefri vardag är att lägga ansvaret på någon annan. En annan lösning som slipper skyffla problemet till en annan människa, är att lägga ansvaret hos en dator. Datorer är, i alla fall i dagsläget, väldigt effektiva på att lösa specifika problem. Beräkningar på tal som skulle kräva flera hundra människors minne för att bevaras effektivt, kan utföras på mikrosekunder hos en dator. Utnyttjandet av denna kraft beror dock på om vi kan specificera problemet på ett betydelsefullt sätt till datorn. En algoritm är, inom kontexten av datorberäkningar, därför just detta. En specifikation på hur ett problem kan lösas ur synvinkeln av en dator.

Kod är hur vi formulerar problem som sedan tolkas på ett förutbestämt sätt till mer maskinnära instruktioner, såsom läsandet och skrivandet av värden i datorns RAM-minne [1]. Optimala sättet att strukturera och tolka kod är något som ofta är knutet till problemet man vill lösa, och därför har olika s.k programmeringsspråk, här ovan Python och nedan Java, paradigmer och kodbibliotek utvecklats [2].

```python
def insertionSort(alist):
    for index in range(1,len(alist)):

        currentvalue = alist[index]
        position = index

        while position>0 and alist[position-1]>currentvalue:
            alist[position]=alist[position-1]
            position = position-1

        alist[position]=currentvalue
```

Figure 1: Exempel på en välkänd sorteringsalgorithm i python

```
DriverListenerAdapter driverListenerAdapter() { return driverListener; }

void lingerResource(final ManagedResource managedResource)
{
    managedResource.timeOfLastStateChange(nanoClock.nanoTime());
    lingeringResources.add(managedResource);
}

boolean isPublicationConnected(final long timeOfLastStatusMessage)
{
    return epochClock.time() <= (timeOfLastStatusMessage + publicationConnectionTimeoutMs);
}

UnavailableImageHandler unavailableImageHandler() { return unavailableImageHandler; }

private void checkDriverHeartbeat()
{
    final long now = epochClock.time();
    final long currentDriverKeepaliveTime = driverProxy.timeOfLastDriverKeepalive();

    if (driverActive && (now > (currentDriverKeepaliveTime + driverTimeoutMs)))
    {
        driverActive = false;

        final String msg = "Driver has been inactive for over " + driverTimeoutMs + "ms";
        errorHandler.onError(new DriverTimeoutException(msg));
    }
}

private void verifyDriverIsActive()
{
    if (!driverActive)
    {
        throw new DriverTimeoutException("Driver is inactive");
    }
}

private int doWork(final long correlationId, final String expectedChannel)
{
    int workCount = 0;

    try
```

Figure 2: Urdrag ur kodbilbioteket Aeron byggt i java för bl.a UDP multicast

1.1 Hur datorer ser på världen

Eftersom datorer i grund och botten består av s.k logic gates, är kod ett sätt att abstrahera bort hårdvaru detaljer för att kunna resonera kring vilka steg en algoritm utför på en högre nivå [3]. Ett exempel från matematiken är derivation och integrering. Vi använder oss av symbol-kombinationer såsom $\int ydx$ och $\frac{dy}{dx}$ för att representera de operationer, som i fallet av polynom addition resp subtraktion av exponentens värde, bör utföras för att uppnå ett önskat resultat. Man är ej explicit med hur det bör gå till, antingen i mån av tid eller textutrymme. Inom datavetenskapen är tankesättet väldigt liknande, men kring hur datorer fungerar. Exempelvis så är en Turingmaskin[4] en teoretisk modell över en generell dator med oändlig minnesstorlek, och kan därför användas som en konceptuell testbädd för att kartlägga datorers gränser. Man kan även härleda diverse egenskaper hos ett givet programmeringsspråk, och därmed dess kapacitet för att lösa vissa uppgifter.

1.2 Handelsmannen som reste

Trots att moderna datorer kan göra beräkningar i en hastighet som överträffar de mest snabbtänkta av oss med en betydlig marginal, har även de en gräns. Denna gräns varierar dock med beräkningarnas natur [5]. Om vi frågar en dator vad summan av två tal är, kan vi rimligtvis ge den tal med flera tusen siffror inom sig, utan att beräkningstiden märks. Då det inom de flesta populära programmeringsspråk finns inbyggda funktioner för addition, blir koden för detta även ganska kort. Här nedan följer ett exempel i python på hur detta skulle kunna gå till:

```python
def adding_exponential_expressions():
    print((10**5000) + (10**3000))
```

Figure 3: Kod i python som leder till utförandet av additionen $10^{5000} + 10^{3000}$

Sådana operationer är relativt vanliga, och instruktionerna också relativt maskinnära, då datorer i stort sett är gjorda för detta[6]. Ökar vi mängden siffror så blir beräkningstiden långsamt större, men siffrorna behöver vara väldigt många innan det blir märkbart.

Vill vi fråga en dator hur vi snabbast tar oss med tunnelbana från Västra Skogen till Östermalmstorg blir det dock betydligt mer komplicerat. Initiala problemet är hur vi på ett konkret sätt kan specificera, dvs koda, problemet för datorn. Tack vare Python's abstraktionsnivå, kan vi beskriva problemet på en hög nivå, med hjälp av klasser, funktioner och variabler, och slipper oroa oss för hur datorn senare tolkar det hela till ettor och nollor.

```python
class Graph:
    def __init__(self):
        self.nodes = set()
        self.edges = defaultdict(list)
        self.distances = {}

    def add_node(self, value):
        self.nodes.add(value)

    def add_edge(self, from_node, to_node, distance):
        self.edges[from_node].append(to_node)
        self.edges[to_node].append(from_node)
        self.distances[(from_node, to_node)] = distance


def dijsktra(graph, initial):
    visited = {initial: 0}
    path = {}

    nodes = set(graph.nodes)

    while nodes:
        min_node = None
        for node in nodes:
            if node in visited:
                if min_node is None:
                    min_node = node
                elif visited[node] < visited[min_node]:
                    min_node = node

        if min_node is None:
            break

        nodes.remove(min_node)
        current_weight = visited[min_node]

        for edge in graph.edges[min_node]:
            weight = current_weight + graph.distance[(min_node, edge)]
            if edge not in visited or weight < visited[edge]:
                visited[edge] = weight
                path[edge] = min_node

return visited, path
```

Figure 4: Djikstras algoritm skriven i python. En välkänd algoritm gjord för att hitta kortaste vägen mellan noder i en graf

Detta problem har en betydligt högre nivå av komplexitet då skaparen av den på figure 4. avbildade algoritmen, en professor inom datavetenskap vid namn Edsger W. Dijkstra, använde sig av grafteoretiska kunskaper för att uppnå denna lösning. Hur ska man då mäta ökningen i svårighet med detta problem? Inom additions problemet är det naturligt att kolla på mängden siffror inblandade. Motsvarigheten inom tåg problemet kan på samma vis pekas till mängden hållplatser. Ser vi dom som noder på en graf, vars kanter associeras med vikter som motsvarar restiderna mellan varandra, kan algoritmen ovan lösa problemet. Finns endast två st, dvs rak väg, blir det optimala valet självklart. Och stegar man sig igenom hur den förnämnda algoritmen hade löst det, ser man också att mängden genomkörningar per rad ej behöver vara så många. Den stora skillnaden här är dock att om man mäter skillnaderna i beräkningstid med ökningar i steg av 1,2 eller 3 hållplatser, så märks en drastiskt mycket högre ökning per steg.

1.3 Beräkningskomplexitet

Det visar sig att beräkningstiden hos algoritmer för olika problem, trots deras olikhet vid ytan, kan generaliseras till gemensamma klasser. En vanligt förekommande metod för klassificering är s.k Big-O notation[7]. Där anses de mest grundläggande operationer som konstanta i tid och minnesförbrukning, och istället så fokuserar man på hur ofta dessa operationer repeteras.

1.4 Artificiell Intelligens

Många tänker troligtvis på Skynet och dess mördarrobot arme från terminator filmserien när dom hör ordet artificiell intelligens, men även om verklighetens motsvarighet som appliceras och utvecklas både industriellt och inom den akademiska världen ej hittills haft några planer på världsdominans, har nivån på uppgifterna som kan utföras och, parallellt med detta, mängden näringsområden som utnyttjar teknologin ökat kraftigt. Vilka youtube videos som rekommenderas till oss [8], hur bra Siri eller Alexa förstår oss [9], och hur träffsäkra våra google resultat är styrs till stor del i dagsläget av artificiell intelligens. Detta pga återupplivandet och vidareutvecklingen av konnektionismen, dvs artificiella neuronnät. YouTubes rekommendations algoritm är ett av de mest genomskinliga och igenkännbara exempel på detta, även om abstraktionsnivån är hög.

Denna rörelse startade dock redan vid år 1943, då Warren McCulloch och Walter Pits för första gången föreslog en model av neuronnät baserad på matematik och algoritmer som kunde utnyttjas av en dator. Dock hade de ej lika kraftulla processorer eller grafikkort på den tiden som nu, vilket i sin tur ledde till vad man idag refererar till som the “AI-winter”, en period av lite till ingen ekonomiskt eller akamediskt intresse inom fältet pga kritiken kring dess brist på praktiskt användande [10].

2 Syfte och frågeställning

Arbetet är en undersökning på hur beräkningskomplexiteten hos olika problem inom datavetenskapen påverkar träffsäkerheten och körningstiden hos problemens respektive algoritmer, och kopplingar mellan dessa problem och teknologiska utmaningar i dagens samhälle. För att konkretisera det hela utförs därför två experiment. Det första jämför effektiviteten hos två olika algoritmer för sortering av en lista innehållande en varierande mängd nummer. Det andra är ett försök att kartlägga förållandet mellan djup, datamängd och precision hos ett artificiellt neuronnät.

Inspiration:

3 Utförande och metod

Mjukvara:

- Python 3.7.3 interpreterare(via anaconda 3.7 distribution) och följande externa kodbibliotek:
  - Algorithms av Keon (https://github.com/keon/algorithms)
  - Matplotlib, OSS startat av John D. Hunter (medföljande i anaconda)
  - Pandas, OSS startat av Wes McKinney (medföljande i anaconda)
  - Scikit Learn, OSS startat av David Cournapeau (medföljande i anaconda)
- Sublime text 3
- Git
- Anaconda Navigator

Hårdvara:

- Apple MacBook Pro 15" från 2018 med följande komponenter:
  - Processor: 2.6 GHz Intel Core i7
  - Minne: 16 GB 2400 MHz DDR4
  - Hårddisk: Macintosh HD
  - Grafikkort:
    - Radeon Pro 560X 4096 MB
    - Intel UHD Graphics 630 1536 MB

DataKällor:

- Kaggle Datasets (https://www.kaggle.com/lava18/google-play-store-apps)

Först skrevs ett document med krav som koden för varje experiment behövde uppfylla. Sedan inhämtades information gällande källor för data att passera genom algoritmerna som skulle prövas. Nödvändiga mjukvarubibliotek laddades ner, och koden skrevs enligt den tidigare skrivna specifikationen.

4 Resultat

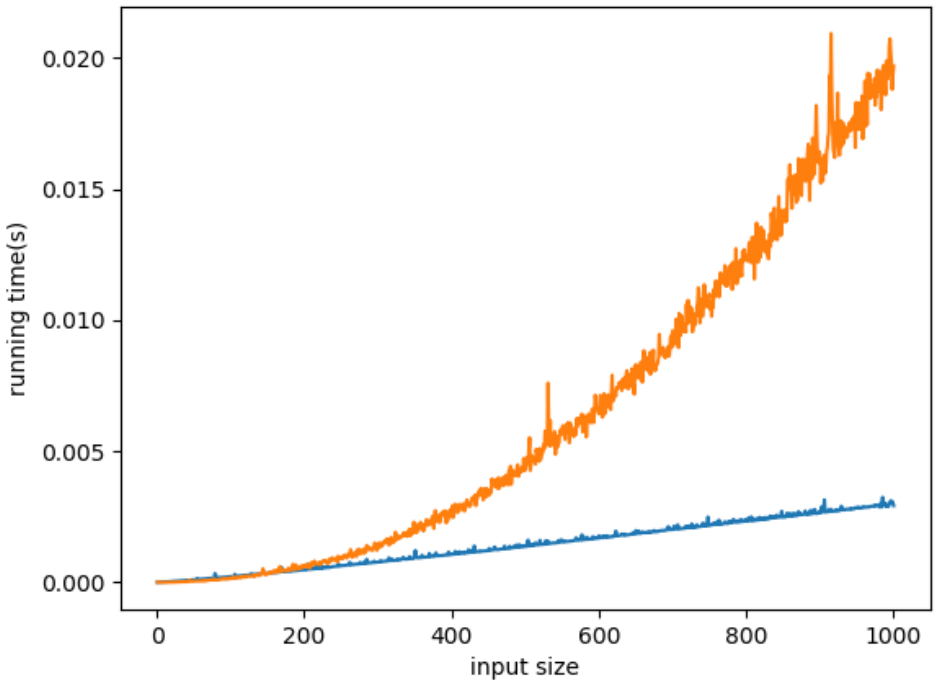

Figure 5: Körningstiden för två olika sorteringsalgoritmer på en lista av 0-1000 st slumpmässiga nummer: Merge sort i blått och Insertion sort i orange (källkod bakom denna och följande grafer hittas vid [11])

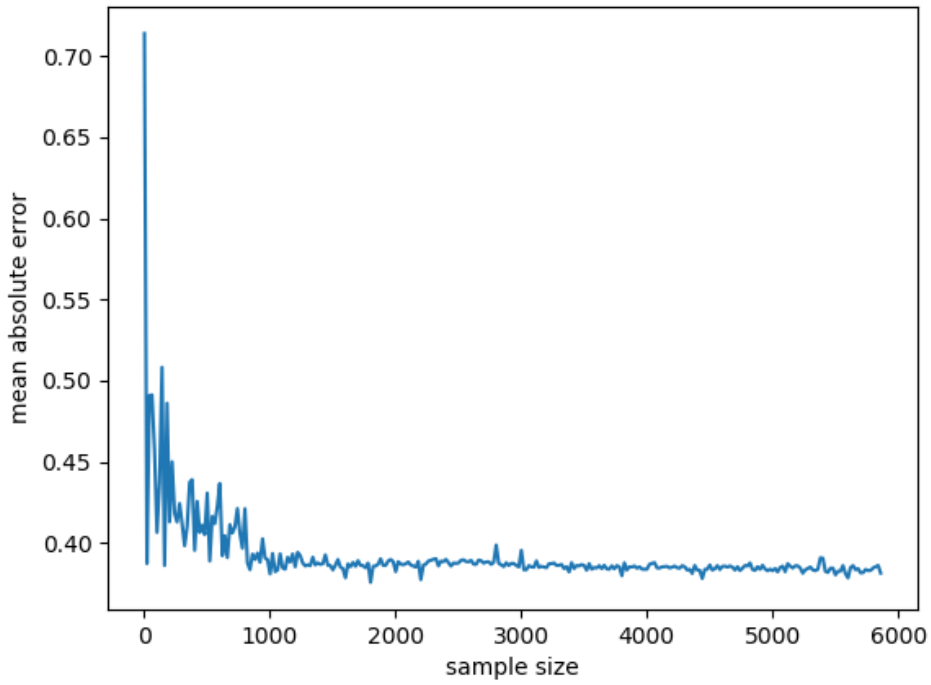

Figure 6: Träffsäkerheten hos ett MLP neuron nät i att förutsäga användarbetyget(0.0-5.0) hos en slumpmässig applikation på Google Play Store baserat på dess filstorlek, antal recensioner och versionnummer efter träning på 0-6000 applikationer

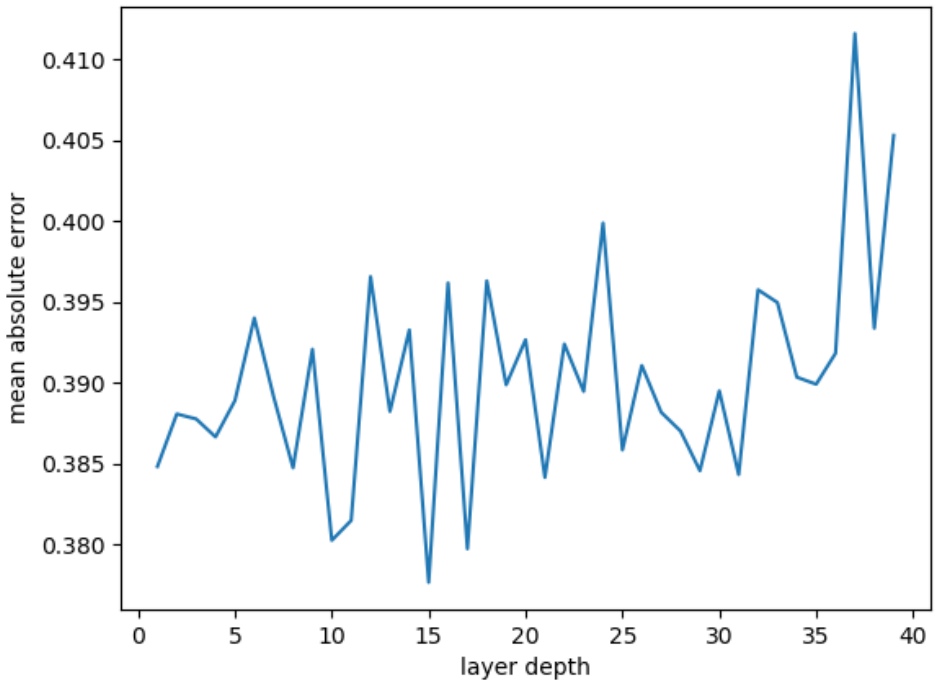

Figure 7: Samma som tidigare dock med en fixerad provstorlek på 1600, och istället ett varierande djup hos nätverkets gömmda lager. 0-40 st, 100 noder per lager

5 Diskussion

På grafen över sorteringsalgoritmerna's körningstid, dvs Figure 4, ser vi att storleken på listorna som ska sorteras påverkar körningstiden hos algoritmerna olika. Insertion sort, avbildat i blått, verkar ha ett nästintill kvadratiskt förhållande till listans storlek, medan Merge sort algoritmen i blått verkar vara linjärt. Gör vi en komplexitets analys av Insertion sort algoritmens kod som syns på Figure 1 ser man att dess yttre for-loop beror på listans längd, medans dess inre while-loop beror på hur nära listan är till att vara sorterad i motsatt riktning. Dvs störst snarare än minst nummer på position 0. Om vi vill bestämma tids komplexiteten hos algoritmen i dess värsta fall i stil med Big-O notation, kan vi använda oss av det faktum att antalet iterationer hos den inre while-loop'en följer en aritmetisk talföljd med differensen 1, om listan är omvänt sorterad. Följdens element bestäms av den yttre for-loop'en, och kan därför summeras via formeln $\frac{n(1+n)}{2}$, där n är listans längd. Då man enligt RAM-modellen[12] anser variabel tilldelning och läsning från minne som operationer med konstant tid och minnes komplexitet, kan algoritmens tidskomplexitet i dess värsta fall generaliseras till $\Theta(n^2)$. Detta då konstanter ignoreras, och den dominerande termen i uttrycket för mängden iterationer var just n^2. Dock är dess minneskomplexitet $O(1)$, då mängden minne som algoritmen kräver ej varierar med listans storlek. Denna analys stämmer väl överens med det förhållande som observeras på grafen. Hade vi gjort en liknande analys för Merge sort, hade vi kommit fram till en tidskomplexitet i värsta fall på $\Theta(nlogn)$, dock med en minneskomplexitet på $O(n)$. Analys av denna algoritm är dock mer invecklad.

På Figure 5 ser man hur viktigt mängden träningsdata är för stabilisering av träffsäkerheten hos ett neuronnät. Även om platån för just denna arkitektur uppnådes vid ca 1600 träningsprov. Då en flerskiktsperceptron användes, känd på engelska som "Multilayer perceptron", finns det olika justeringar som skulle kunna göras för att förbättra dess prestanda vidare, kända under samlingsnamnet "hyperparameter optimization"[13]. Det finns metodiker för att utföra detta manuellt, men även på ett automatiserat sätt med hjälp av diverse algoritmer. Ett manuellt försök till detta avbildas i Figure 6, då varierande djup hos neuronnätets gömda lager testades på en fixerad mängd träningsprov, utan några statistiskt betydelsefulla förbättringar under de ca 5 gånger varje lagerdjup testats. Dock hade optimisering på algoritmisk väg möjligtvis kunnat hitta både ett mer fördelaktigt djup än de som testats, men också bättre värden hos de andra justerbara parametrarna inom nätverkets struktur.

Det första experimentet illustrerar en röd tråd som går igenom dom båda, men även samhället i stort. Även om vi tack vare moore's lag har möjligheten att expandera x-axeln ävsevärt innan mängden tid som krävs för sortering blir praktiskt ohanterbar, finns det uppenbara gränser även idag. En algoritm med en tidskomplexitet inom $O(n^2)$ leder till körningstider på uppemot 31.7 år vid en inmatningsstorlek på 1 miljard, givet att datorn den kör på även kan genomföra samma algoritm på $10\,\mu s$ given en inmatningsstorlek på 1000. Detta i kontrast med att den MacBook som experimentet körde på endast kunde hinna slutföra algoritmen på ca 20 ms, dvs ca 500 gånger långsammare, kombinerat med det faktum att facebook har över 2 miljarder aktiva användare, blir det uppenbart att även moderna datorer som är prestandamässigt överlägsna de mest kraftulla superdatorer från endast några tiotal år sedan är vissa problem fortfarande långt utom räckhåll.

I vissa fall går problemen att distribuera över en rimlig mängd datorer för att övervinna detta, men i fallet av algoritmer med en tidskomplexitet inom $O(a^n)$ eller till och med $O(n!)$ blir det ej rimligt även om man haft tillgång till en stor mängd datorer vid gränserna av vad både fysiken och datavetenskapen tillåter. Vad vi kan och inte kan gjora med datorer vare sig om det är självkörande bilar, rekommendationer för vilka låtar vi ska lyssna på eller robotar som kan skapa poesi, kokar därför ner till om problemen vi i grunden försöker lösa har en algoritm inom en komplexitetsklass vars gräns för hanterbara inmatningsstorlekar är tillräckligt låg för att dagens datorer, eller de som kan skapas inom en rimlig framtid, kan genomföra den. Detta presenterar flera utmaningar som ej har någon garanti på att kunna resolveras. Även om vi inte hittat en algoritm inom en lämplig komplexitetsklass för ett givet problem, är det inte nödvändigtvis så att någon sådan ej existerar. Även inom de mest avlägsna av komplexitetsklasser, är det i dagsläget en öppen forskningsfråga kring exakt hur vissa av dessa relaterar till varandra, och om det möjligtvis finns någon algoritm som vid dess upptäckt, löser ett givet problem som även uppstår i bl.a olika transport, kryptering och schemaläggnings problem runt om i samhället. Dock är det väldigt svårt att säga hur stort av ett hinder detta blir i verkligheten, då diverse heuristiker lett till algoritmer med rimliga körningstider och tillfredställande träffsäkerhet, trots att dess problem tillhört komplexitetsklasser som forskare länge ansett som tecken på att ge up.

References

[1] Kahanwal Brijender. Abstraction level taxonomy of programming language frameworks. *International Journal of Programming Languages and Applications (IJPLA)*, 3:1–3, 2013.

[2] Sebastian Nanz and Carlo A. Furia. A Comparative Study of Programming Languages in Rosetta Code. *arXiv e-prints*, pages 1–11, Aug 2014.

[3] Bradley N. Miller and David L. Ranum. *Problem solving with algorithms and data structures using python*, chapter 1, pages 34–47. Jim Leisy, first edition, 2006.

[4] Michael Sipser. *Introduction to the theory of computation*, chapter 3, pages 137–153. Thomson Course Technology, second edition, 2005.

[5] J. Hartmanis and R. E. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

[6] Clive Maxfield and Alan Chew. *The Definitive Guide to How Computers Do Math*, chapter 4, pages 70–123. John Wiley I& Sons, first edition, 2005.

[7] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*, chapter 1, pages 3–14. Cambridge University Press, first edition, 2009.

[8] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems*, New York, NY, USA, 2016.

[9] Grant P. Strimel, Kanthashree Mysore Sathyendra, and Stanislav Peshterliev. Statistical Model Compression for Small-Footprint Natural Language Understanding. *arXiv e-prints*, Jul 2018.

[10] Sir James Lighthill. Artificial intelligence: A general survey. *Artificial Intelligence: a paper symposium*, 1972.

[11] Mbwenga Maliti. P v np: En för alla?(källkod). `https://github.com/OpUs-Nebula/P-v-NP-En-f-r-alla-.git`, 2019.

[12] Department of Computing Science Umeå Universitet. The ram model of computation. `https://www8.cs.umu.se/kurser/TDBA77/VT06/algorithms/BOOK/BOOK/NODE12.HTM`, Jun 1997.

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*, chapter 5, pages 120–121. The MIT Press, 2015.