MALMÖ UNIVERSITET

Teknik och samhälle
Datavetenskap och medieteknik

Examensarbete
15 högskolepoäng, grundnivå

Praktisk jämförelse av strängsökningsalgoritmer

Empirical comparison on string-search algorithms

Ekaterina Korotetskaya

Examen: kandidatexamen 180 hp
Huvudområde: datavetenskap
Program: datavetenskap och applikationsutveckling
Datum för slutseminarium: 2021-08-31

Handledare: Jesper Larsson
Examinator: Olle Lindeberg

Sammanfattning

Strängsökning användas i stor utsträckning i bioinformatik, IT-säkerhet (nätverksintrång-detektion) och för sökning av en söksträng i en text mm, därför utvecklades senaste åren stort antal söksträng algoritmer. Denna uppsats handlar om praktisk implementering av följande stängsökningsalgoritmer: Brute-force, Boyer-Moore och Knuth-Morris-Pratt algoritmer med syftet att jämföra hur algoritmens effektivitet påverkas av typ av data och storlek på söksträng. Mått på algoritmernas effektivitet är antal jämförelser som algoritmerna utför. Det är tre typer av text som används för tester: slumpmässigt genererad text som innehåller ettor och nollor som tecken, text som representerar DNA sekvens som består av ett alfabet med fyra bokstäver och en text som innehåller tecken från engelska alfabetet. Resultatet visar hur algoritmernas effektivitet ändras med olika typ av data och förklarar vad i algoritmernas operationssätt som ligger bakom de uppmätta resultaten.

Abstract

String search is widely used in bioinformatics, IT security (network intrusion detection) and for search of a string in a text etc. This project deals with the practical implementation of the following string-search algorithms: Brute-force, Boyer-Moore and Knuth-Morris-Pratt algorithms with the aim of comparing how the algorithm's efficiency is depending on the typ of the dataset and the search string size. A measure of the algorithm's efficiency is the number of comparisons performed by the algorithm. There are three types of text used for tests: random text (0 and 1 as characters), text that represents a DNA sequence that consists of an alphabet of four letters and a text that contains characters from the English alphabet. This project makes an evaluation of the algorithm's result depending on the type of data where the search is performed and depending on the size of search strings.

Innehåll

1 Inledning **1**
- 1.1 Bakgrund och tidigare forskning . . . 1
- 1.2 Frågeställning . . . 2

2 Litteraturstudie **2**
- 2.1 Data . . . 2
- 2.2 Genomförande och resultat . . . 4

3 Metod **6**
- 3.1 Experiment . . . 6
- 3.2 Förberedelser . . . 6
- 3.3 Genomförande . . . 6
  - 3.3.1 Effektivitetsmätning . . . 7
  - 3.3.2 Testmiljö . . . 7
  - 3.3.3 Avgränsningar . . . 7

4 Strängsökningsalgoritmer **9**
- 4.1 Brute-force algoritm . . . 9
- 4.2 Boyer-Moore algoritm . . . 10
- 4.3 Knuth-Morris-Pratt algoritm . . . 13
- 4.4 Sammanfattning . . . 16

5 Resultat **17**
- 5.1 Slumpmässig binär sekvens . . . 17
- 5.2 DNA sekvens . . . 18
- 5.3 Engelsk text . . . 18

6 Analys och Diskussion **19**
- 6.1 Brute-force algoritm . . . 19
  - 6.1.1 Generalisering . . . 20
- 6.2 Boyer-Moore algoritm . . . 20
  - 6.2.1 Korta strängar . . . 20
  - 6.2.2 Medelstora strängar . . . 21
  - 6.2.3 Långa strängar . . . 21
  - 6.2.4 Generalisering . . . 22
- 6.3 Knuth-Morris-Pratt algoritm . . . 23
  - 6.3.1 Generalisering . . . 23

7 Slutsatser och vidare forskning **24**

8 Referenser **25**

1 Inledning

1.1 Bakgrund och tidigare forskning

En sträng är en sekvens av tecken som kan bestå av bokstäver, siffror och specialtecken som representeras av någon typ av teckenkodning. De vanligaste är utökad ASCII som är 8-bitars teckenkodning [1]. Strängar användas till olika syften. För att dela information och kommunikation används strängar i form av textmeddelande där flera strängar kan bygga en text. Inom datalogin och vissa programmeringsspråk hanteras strängar som en datatyp. Inom biologi representerar strängar DNA sekvens.
Strängsökning kan göras i en text på till exempel en webbsida eller text editor. Strängsökning användas också i stor utsträckning i bioinformatik, IT-säkerhet (nätverksintrångdetektion). I bioinformatik används strängsökning för att hitta ett visst mönster i DNA sekvenser. DNA är genetisk material som innehåller fyra kvävebaser: adenin (A), guanin (G), cytosin (C) och tymin (T) och de första bokstäverna i deras kemiska namn A, G, C och T bygger en text där sökningen genomförs. En bra algoritm för strängsökning i DNA sekvenser är viktigt pga kontinuerligt ökande storlek av DNA databaser [6], mängden av biologisk data fördubblas var 20:e månad [15]. Antal cyberattacker som utförs med syfte att få tillgång till känslig information har ökat de senaste åren. För att upptäcka ett intrång kan granskning av paket-headers för varje inkommande paket göras [8]. Paket-headers söks genom för att hitta misstänkta paket för att kunna hantera detta [8]. Snabba algoritmer för strängsökningen är viktiga eftersom effektivitet av system som upptäcker intrång på nätverk beror på algoritmers hastighet. Strängsökning genomförs också i binär data då söks efter en sträng som representeras av ettor och nollor som tecken. Detta är viktigt vid till exempel hantering av komprimerade texter eller kommunikation när data skickas.
Strängsökningsalgoritmer kan delas i två grupper: algoritmer för exakt strängsökning och approximerad [4]. Vid exakt strängsökning strängsökningsalgoritmerna hittar bara träffar som är exakt som söksträngen, dvs söksträngen måste helt matha datan [4]. Vid approximerad strängsökning letar algoritmerna efter en söksträng men hittar även träffar som inte är exakt som söksträngen och den här typen av strängsökning är användbar om det finns möjlighet till stavningsfel i söksträngen [4]. Algoritmerna kan söka efter bara den första träffen och avbrytas därefter eller algoritmerna kan leta efter alla träffar av en söksträng i datan. Algoritmerna kan också definiera en träff genom att räkna överlappande söksträngar som separata träff eller inte. Ett exempel är texten “AAA” och söksträngen “AA”. Det kan räknas som om det finns en eller två träffar.
I vetenskapliga artiklar beskrivs ofta teoretisk utvärdering av algoritmer. Vid teoretisk utvärdering beskrivs hur snabb en algoritm är men i praktiken algoritmens resultat varierar beroende på olika typer av indata. De lästa artiklar som utvärderar algoritmers beteende praktisk använder inte stora söksträngar. Pga att inga utvärderingar kan göras under exakt samma omständigheter kan det vara intressant med experiment med korta söksträngar för att jämföra resultatet med föregående forskning och med stora söksträngar för att detta inte finns dokumenterat.
Målet med arbetet är att praktisk jämföra algoritmernas effektivitet beroende på mängden av data, längden av söksträngar och olika typer av alfabet. Med varierande storlek på söksträngar och olika typ av data kan algoritmernas styrkor och svagheter bedömas.

1.2 Frågeställning

-Hur söksträngsstorlek och datatyp påverkar strängsökningsalgoritmers effektivitet?
De tre typerna av data är:

- Slumpmässigt skapad binär sekvens
- DNA sekvens
- Engelsk text

2 Litteraturstudie

Detta avsnitt beskriver tidigare utförd forskning inom strängsökning och avsnittet är indelat i två delar: den första beskriver datan som har används i föregående forskningen och i den andra beskrivs resultat av föregående forskningen.

2.1 Data

I de lästa vetenskapliga artiklarna görs jämförelser av olika strängsökningsalgoritmer och för detta används korta söksträngar som indata. Typer av data är bland annat: slumpmässig binär sekvens (1 och 0 som tecken), DNA sekvens och Engelsk text. Tabell 1 visar en sammanfattning av litteraturöversikten över olika typer av text som används i olika vetenskapliga studier, olika datastorlek och söksträngsstorlek.

Typ av data	**Artiklar, datakälla**	**Sökträngsstorlek**	**Datastorlek**
Slumpmässigt binär sekvens	[**7**] Slumpmässigt sekvens av 0 och 1	[**7**] 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 60, 80 och 100 tecken	[**7**] 150000 tecken
	[**13**]Slumpmässigt sekvens av 0 och 1	[**13**]sökträngsstorlek är alla heltal från 1 till 14 tecken (dvs 1, 2 ... 14)	[**13**] 10000 tecken
DNA sekvens	[**7**] Datan är från GenBank DNA-databasen	[**7**] 10, 20, 30, 40, 50 och 100 tecken	[**7**] 997642 tecken
	[**12**] DNA-text skapad slumpmässigt	[**12**]sökträngsstorlek är alla heltal från 2 till 20 (dvs 2, 3 ... 20)	[**12**] 40000 tecken
	[**15**] Data har hämtats från National Centre of Biotechnology Information (NCBI)	[**15**]sökträngar från 2 till 20 med två mellan varje storlek (2, 4, 6...20)	[**15**] 837 gensekvenser som innehåller cirka 2,5 miljarder tecken med ett alfabet bestående av fyra tecken (GCTA)
Engelsk text	[**2**] Texten genererades slumpmässigt från 25000 ord från UNIX English Dictionary	[**2**] sökträngsstorlek är alla heltal från 2 till 22 (dvs 2, 3, ...22)	[**2**] 200000 tecken
	[**7**] Textkällan är ospecificerad (text från en webbsida)	[**7**] 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 60, 80 och 100 tecken	[**7**] 148188 tecken
	[**12**] Textkällan är ospecificerad	[**12**]sökträngsstorlek är alla heltal från 2 till 20 (dvs 2, 3, ...20)	[**12**] 48000 tecken
	[**13**] Textkällan är ospecificerad	[**13**]sökträngsstorlek är alla heltal från 1 till 14 tecken (dvs 1, 2 ... 14)	[**13**] 10000 tecken

Tabell 1: Datainformation i lästa artiklar.

2.2 Genomförande och resultat

[**2**] under experimenten undersöktes olika strängsökningsalgoritmer och bland annat Brute-force, Boyer-Moore och Knuth-Morris-Pratt algoritmerna. Text med storleken 200000 tecken skapades slumpmässigt av 25000 ord från UNIX Engelsk Ordbok och alla ord separeras med ett mellanslag tecken. Det genomförs sökning efter varje ord från UNIX Engelsk Ordbok. Under experimentet uppskattades algoritmernas effektivitet genom att räkna genomsnittligt antal jämförelser och mäta exekveringstid för sökningen.
Resultatet av denna undersökningen visade att för Engelsk text gör Brute-force algoritmen flest antal jämförelser medan Knuth-Morris-Pratt algoritm gör nästan lika många jämförelser som Brute-force. Boyer-Moore algoritmen gör sex gånger mindre antal jämförelser än Knuth-Morris-Pratt och Brute-force algoritmerna i genomsnitt. För alla tre algoritmerna går det inte att dra någon slutsats hur antal jämförelser ändras med ökningen av sökströngslängden pga att de uppmätta antal jämförelser visar inte någon tydlig trend.

[**7**] under experimentet jämfördes effektiviteten hos Brute Force, Knuth-Morris-Pratt och Boyer-Moore algoritmerna. För Engelsk data är textkällan ospecificerad och alfabetets storlek är 70 tecken. Koden exekveras 50 gånger för varje sökströngslängd. Ett mått på algoritmernas effektivitet är förhållandet mellan antal teckenjämförelser och antal tecken passerade i textströngen (dvs antal teckenjämförelser dividerad med antal tecken passerade i textströngen). Antal tecken passerade i textströngen är $n - m + 1$ (där n är textlängd och m är sökströngslängd) eftersom algoritmerna är designade att hitta alla träffar av en sökström.
Resultatet visade att för Engelsk text är Boyer-Moore algoritmen mest effektiv pga att förhållande mellan antal teckenjämförelser och antal tecken passerade i textströngen är 0,172907 medan Knuth-Morris-Pratt och Brute-force algoritmerna 1,028437 respektive 1,032923.
Resultatet är samma för DNA sekvens och binär sekvens: Boyer-Moore algoritmen är mer effektiv än Knuth-Morris-Pratt och Brute-force algoritmerna.
Brute-force algoritmen är mer effektiv för Engelsk text än för texter med korta alfabet. Boyer-Moore algoritmen är mer effektiv för Engelsk text, mindre effektiv för DNA sekvens och slumpmässig binär sekvens. Knuth-Morris-Pratt algoritmen gör i genomsnitt nästan samma antal jämförelser för alla tre typer av datan.

[**12**] under experimenten undersöktes Brute-force, Boyer-Moore och Knuth-Morris-Pratt algoritmerna och algoritmernas effektivitet uppskattades genom att mäta exekveringstid av sökningen. Det söks efter 1000 olika söksträngar med storlekar mellan 2 till 20 och koden exekveras 100 gånger för varje söksträng. Data och söksträngar som representerar DNA skapades slumpmässigt och består av fyra tecken (A,G, T och C). För Engelsk data är textkällan ospecificerad och alfabetets storlek är 32 tecken. Resultatet visar att Boyer-Moore algoritmen är mer effektiv för Engelsk text och slumpmässigt skapad DNA sekvens. Exekveringstid hos Boyer-Moore algoritmen minskar med ökningen av sökströngslängden. Exekveringstiden för Knuth-Morris-Pratt algoritmen ändras inte mycket med ändringen av sökströngslängden. Brute Force algoritmen har längre exekveringstiden än de två andra algoritmerna.

[13] under experimentet undersöktes bara Boyer-Moore algoritmen. Datatyp är bland annat Engelsk text och slumpmässigt skapad binär sekvens. För strängsökningen används 300 slumpmässigt valda söksträngar. Under experimentet mättes algoritmernas effektivitet genom att mäta antalet accesser till strängar och totala antalet exekverade maskininstruktioner, kostnaden för förbehandling ignoreras. Resultatet visar att med ökningen av söksträngslängden minskade genomsnittliga antalet accesser till strängar per tecken i båda två typer av texter. Boyer-Moore algoritmen visar bättre resultat för Engelsk text än för binär sekvens. För stort alfabet och lång sökträng exekverar algoritmen mindre än 1 instruktion per ett tecken [13].

[15] under experimentet jämförs effektiviteten hos bland annat Brute-force, Knuth-Morris-Pratt och Boyer-Moore algoritmerna. För varje söksträngslängd används 50 slumpmässigt valda söksträngar och genomsnittlig exekveringstid mäts. Resultatet visar att Boyer-Moore algoritmen har bättre resultat (tar mindre tid) än Brute-force och Knuth-Morris-Pratt algoritmerna i alla testfall. Brute-force algoritmen tar längst tid. För alla tre algoritmerna går det inte att dra några slutsatser om hur antalet jämförelser ändras med ökningen av söksträngslängden pga att de uppmätta antal jämförelser inte visar någon tydlig trend.

3 Metod

3.1 Experiment

Metoden som har valts för examensarbetet är ett experiment. Syftet med experimentet är att göra en praktisk utvärdering av algoritmer och förstå hur algoritmernas effektivitet påverkas beroende på typ av data (data som består av olika typer av alfabet) och storlek på söksträng. Med varierande data och söksträng kan algoritmernas styrkor och svagheter utvärderas. Experiment är lämpligt pga att det ses som den mest vetenskapliga metoden där speciella egenskaper kan isoleras från omgivningen och då kan länkar mellan flera faktorer visas och orsak och verkan undersökas [5].

3.2 Förberedelser

Tre olika typer av data valdes för att jämföra de tre algoritmerna:

- Slumpmässigt genererad binär sekvens
- DNA sekvens
- Engelsk text

Slumpmässigt genererade binär sekvens
Slumpmässigt genererade sekvenser består av två tecken (0 och 1). Data skapades för detta testet som en sträng med längden 5583968 tecken.
DNA sekvens
Den typ av data har valts eftersom sökning efter DNA-sekvenser har ökat senaste tiden pga att mängden genetisk data fördubblas var 20:e månad [15]. Datastorlek av DNA sekvensen som används för experimentet är 5583968 tecken och alfabet storlek är 4 tecken. DNA består av fyra kvävebaser: adenin (A), guanin (G), cytosin (C) och tymin (T) och de första bokstäverna i deras kemiska namn A, G, C och T bygger en text. Källan till datan är boken "Algorithms" (Sedgewick R.) kompletterande material [17]. Innan exekveringen av strängsökningsalgoritmer genomförs inläsning av en rad i taget från en textfil för att skapa en stor sträng med längden 5583968 tecken där sökningen genomförs.
Engelsk text
Den här typen av data har valts pga att engelska är ett av de mesta talade [21] och ett av de vanligaste språken som används på internet [20]. Källan av datan är boken "Algorithms" (Sedgewick R.) kompletterande material [18]. Datastorlek är 5583968 tecken och alfabet storlek är 84 tecken. Innan exekveringen av strängsökningsalgoritmer genomförs inläsning av en rad i taget från en textfil för att skapa en stor sträng där sökningen genomförs.

Innan experimentet genomföras inläses först data från DNA textfil för att skapa en stor sträng med längden 5583968 tecken där sökningen genomförs. Därefter skapas den slumpmässiga binära sekvensen och Engelsk text med samma längd för att jämförelserna ska bli likvärdiga.

3.3 Genomförande

Det skapades slumpmässigt söksträngar med längd som fördubblas från ett tecken till att längden inte går att fördubbla mer (4194304 tecken, 2^{22}). Söksträngar av varje längd skapa-

des 100 gånger och startpunkt för varje sökträng har valts slumpmässigt. Varje sökträng söktes 10 gånger. När experimentet är klar beräknas genomsnittligt antal jämförelser för varje söksträngsstorlek för varje typ av data och varje algoritm. Anledningen att flera sökningar görs med samma söksträngar är för att få ett bra medelvärde.
Den största söksträng som Knuth-Morris-Pratt algoritmen kan hantera på den datorn som experimentet körs på är 1048576 tecken (2^{20}) och vid fördubblingen av den här söksträngen räcker inte datorns arbetsminne. Därför genomförs tester av Knuth-Morris-Pratt algoritmen med söksträngar av längden från 1 (2^0) till 1048576 tecken (2^{20}).

3.3.1 Effektivitetsmätning

För att genomföra kvantitativ dataanalys kommer antal jämförelser som en algoritm genomför att användas. Algoritmerna stannar inte efter första träffen och hur algoritmerna definierar en träff beskrivs i del 3.3.3 Avgränsningar.
Varje algoritm har en räknare för att beräkna antalet jämförelser genom att räkna antalet teckenjämförelser som gjordes mellan datan och söksträngar. När mätningarna är klara beräknas genomsnittligt antal jämförelser. Antal jämförelser som en algoritm genomför är ett viktigt mått för att bedöma algoritmens effektivitet. Exekveringstid har inte valts som ett mått på algoritmernas effektivitet eftersom det är ett osäkert mått pga att det kan bero på kodens struktur, datorns kapacitet och tillgängliga resurser.

3.3.2 Testmiljö

Alla algoritmerna implementeras i programmeringsspråk Java och exekveras på en dator med processor Intel® Core™ i5-7200U CPU @ 2.50GHz × 4, med operativsystem Ubuntu 20.10. Java version = 1.8.

3.3.3 Avgränsningar

De senaste 40 åren har mer än 120 algoritmer för strängsökning utvecklats [3]. Det finns olika sökningsstrategier och för att begränsa området och omfattning av tester är fokus på de tre algoritmerna:

- Brute-force algoritm
- Boyer-Moore algoritm
- Knuth-Morris-Pratt algoritm

Brute-force, Boyer-Moore och Knuth-Morris-Pratt algoritmerna som var tagna från boken "Algorithms" [1] har implementerats för att enbart hitta första träffen och efter avbrytas. För experimentet görs det ändringar i de tre nämnda algoritmerna för att de skulle hitta alla träffar av en sökträng, dvs stannar inte efter den första träffen. De tre algoritmerna definierar inte träff på samma sätt. Till exempel i texten "AAA" kommer Brute-force algoritmen hitta två träffar av söksträngen "AA", medan Knuth-Morris-Pratt och Boyer-Moore algoritmerna kommer hitta bara en träff. För att göra så att alla tre algoritmerna få samma antal sökträffar och för att kunna jämföra de efter experimentet görs ändringar i Brute-force algoritmen så att den hittar lika många träffar som de andra algoritmerna.

Datatyp	**Datastorlek**	**Alfabet storlek**
Slumpmässigt genererad binär sekvens	5583968 tecken	2
DNA sekvens	5583968 tecken	4
Engelsk text	5583968 tecken	84

Tabell 2: Sammanfattning av datan som används för experiment

4 Strängsökningsalgoritmer

För att ge inblick i implementerade algoritmer förklaras de här mer detaljerat.

4.1 Brute-force algoritm

- tidskomplexitet är $O(\text{mn})$ i värsta fall, där n är textlängd och m är söksträngslängd
- förbehandling behövs inte

Brute-force är den enklaste algoritmen i jämförelsen med andra strängsökningsalgoritmer och kräver inte en förbehandling innan själva sökningen. Brute-force algoritmen gör sökning i texten från tecknet på platsen 0 till tecknet på platsen $n - m$, där n är textlängd och m är söksträngslängd. Den här algoritmen börjar jämföra på platsen 0 i både: text och sökträng och fortsätter jämföra åt höger om de två tecken matchar varandra. Om det uppstår en misslyckad jämförelse eller hela söksträngen hittades då förflyttas söksträngen bara ett steg åt höger och jämförelse börjas om från det första tecknet i söksträngen.
I *Algoritm* 1 nedan [1] presenteras koden av Brute-force algoritmen. I början initieras två pekare, en för text (*txtPointer*) och en för sökträng (*patPointer*) som visas på rad 8 och rad 10 i *Algoritm* 1. Så länge de pekar på matchande tecken ökar *patPointer* med ett ($patPointer++$) vid varje träff och jämförs vidare med nästa tecken i texten tills misslyckad jämförelse påträffas eller hela söksträngen är hittad. Vid misslyckad jämförelse avbrytas yttre for-loop (rad 13 i *Algoritm* 1), *txtPointer* ökas med ett och *patPointer* backar till början av söksträngen och jämförelse börjas om.

```
1  public class BruteForce {
2   public static void search(String txt, String pat){
3     int amountComparisons=0;
4     int amountHits=0;
5     int patLength = pat.length();
6     int txtLength = txt.length();
7
8     for (int txtPointer=0; txtPointer<=txt.length-pat.length; txtPointer++) {
9          int patPointer;
10         for (patPointer=0; patPointer<pat.length; patPointer++){
11            amountComparisons+=1;
12            if (txt.charAt(txtPointer+patPointer)!=pat.charAt(patPointer)){
13               break;
14            }
15         }
16         if (patPointer==patLength) {
17              amountHits++;
18         }
19      }
20  }
21
22  public static void main(String[] args) throws IOException {
23      String txt = "ABBEBCDE";
24      String pat = "BCD";
25      search(txt, pat);
```

```
26    }
27  }
```

Algoritm 1. Brute-force algoritm.

Brute-force algoritmen förklaras nedan i Figur 1 med ett exempel, där ABBEBCDE är text och BCD är sökströngen.
Första tecknet i texten (på rad 1 i Figur 1) och första tecknet i sökströngen (på rad 2 i Figur 1) är inte samma (misslyckad jämförelse markerad med röd färg). Därför ska inte jämförelsen fortsätta vidare och sökströngen förflyttas ett steg åt höger.
Andra tecknet i texten (på rad 1 i Figur 1) och första tecknet i sökströngen (på rad 3 i Figur 1) är lika därför fortsätter jämförelse av texten och sökströngen vidare (en träff markerad med grön färg). På grund av att tredje tecknet i texten och andra tecknet i sökströngen inte är samma kommer inte jämförelserna fortsätta vidare och sökströngen förflyttas ett steg åt höger.
På rad 6 i Figur 1 hittas hela sökströngen. Algoritmen är klar pga att längden av resten av texten är mindre än längden av sökströngen.

1	A	B	B	E	B	C	D	E
2	B	C	D					
3		B	C	D				
4			B	C	D			
5				B	C	D		
6					B	C	D	

Figur 1: Grafisk representation av Brute-force algoritmen.

4.2 Boyer-Moore algoritm

- tidskomplexitet för förbehandlingen är $O(m + R)$, där m är sökströngslängd och R är antal tecken i alfabetet [1]
- tidskomplexitet för sökningen är $O(n/m)$ i bästa fall där n är textlängd och m är sökströngslängd och $O(m * n)$ i värsta fall [1]

Boyer-Moore algoritmen, som Knuth-Morris-Pratt algoritmen gör förbehandling av sökströngen innan sökningen kan genomföras. Förbehandling genomförs för att veta hur mycker sökströngen ska förflyttas vid en misslyckad jämförelse. Under förbehandlingen skapas en endimensionell heltal array (som kallas $right[]$ i Tabell 3) med storleken 256 [1]. Varje plats i arrayen representerar en av 256 möjliga tecken vid 8-bitars teckenkodning EXTENDED ASCII och den initieras med värde -1 för varje plats i arrayen. För varje tecken i alfabetet anges i arrayen $right[]$ index av varje teckens sista position i sökströngen. Om tecknet inte finns i sökströngen så står det kvar -1 i arrayen.
I Tabell 3 visas array för ordet “illness” som fås när förbehandlingen är klar. Värdet i

arrayen *right* visar varje teckens sista position i söksträngen. Det här värdet visar hur mycker söksträngen ska förflyttas åt höger vid en misslyckad jämförelse med texten.

	I	**L**	**L**	**N**	**E**	**S**	**S**	**right[]**
A	-1	-1	-1	-1	-1	-1	-1	**-1**
B	-1	-1	-1	-1	-1	-1	-1	**-1**
C	-1	-1	-1	-1	-1	-1	-1	**-1**
D	-1	-1	-1	-1	-1	-1	-1	**-1**
E	-1	-1	-1	-1	**4**	-1	-1	**4**
...	...	...	...	...	...	...	...	...
I	**0**	-1	-1	-1	-1	-1	-1	**0**
...	...	...	...	...	...	...	...	...
L	-1	**1**	**2**	-1	-1	-1	-1	**2**
...	...	...	...	...	...	...	...	...
N	-1	-1	-1	**3**	-1	-1	-1	**3**
...	...	...	...	...	...	...	...	...
S	-1	-1	-1	-1	-1	**5**	**6**	**6**

Tabell 3: Boyer-Moore algoritm, uppbyggnad av array.

Efter förbehandlingen genomförs själva sökningen av söksträngen i texten. Det initieras två pekare: *txtPointer* och *patPointer* (*Algoritm* 2). Både text och söksträng är placerade så att första tecknen är under varandra (Figur 2, rad 1 och 2). Jämförelsen börjar från det sista tecknet i söksträngen, dvs jämförelse av texten och söksträngen genomföras från höger till vänster.

```
1  public class BoyerMoore {
2      private final int R;
3      private static int[] right;
4      private char[] pattern;
5
6      public BoyerMoore(char[] pattern, int R) {
7          this.R = R;
8          this.pattern = new char[pattern.length];
9
```

```
10          //copy  pattern
11          for (int j = 0; j < pattern.length; j++)
12              this.pattern[j] = pattern[j];
13
14          // position of rightmost occurrence of c in the pattern
15          right = new int[R];
16          for (int c = 0; c < R; c++)
17              right[c] = -1;
18
19          for (int j = 0; j < pattern.length; j++) {
20           right[pattern[j]] = j;
21          }
22      }
23
24      public void search(char[] text) {
25       int amountComparisons = 0;
26       int amountHits = 0;
27       int patLength = pattern.length;
28       int txtLength = text.length;
29       int skip;
30
31       for (int txtPointer=0; txtPointer<=txtLength-patLength; txtPointer+=
             skip) {
32           skip = 0;
33           for (int patPointer=patLength-1; patPointer>=0; patPointer--) {
34              if (pattern[patPointer] == text[txtPointer+patPointer]) {
35               amountComparisons++;
36              }
37              else {
38               amountComparisons++;
39               skip=Math.max(1,patPointer-right[text[txtPointer+patPointer]]);
40               break;
41              }
42           }
43           if (skip == 0) {
44              skip = patternLength;
45              amountHits ++;
46           }
47        }
48      }
49
50      public static void main(String[] args){
51          String txt = "arghjwizlpsqfillness";
52          String pat = "illness";
53          char[] pattern = pat.toCharArray();
54          char[] text    = txt.toCharArray();
55          BoyerMoore bm = new BoyerMoore(pattern, 256);
56          bm.search(text);
57      }
58  }
```

Algoritm 2. Boyer-Moore algoritm, sökning efter sträng [1].

Om ett tecken från texten och ett tecken från söksträng matchar varandra fortsätter jämförelser tills en misslyckad jämförelse eller tills hela söksträngen är hittad. Algoritmen letar efter alla träffar av en sträng i texten, dvs algoritmen stannar inte efter första träffen.

Vid en misslyckad jämförelse finns det tre möjligheter [1]:

- Om tecknet i texten där den misslyckade jämförelsen hände finns i söksträngen förflyttas söksträngen åt höger så att tecknet i söksträngen ligger rakt under detta tecken i texten. För att veta om ett tecken finns i söksträngen kontrolleras vilket heltal det motsvarar i arrayen *right* (rad 39 i Algoritm 2). Heltalen i arrayen right visar hur mycket söksträngen ska förflyttas åt höger. I Figur 2 på rad 1-2 visas misslyckad jämförelse vid tecknet I i texten. I Tabell 3 visas att tecknet I motsvarar heltal 0. Det betyder att tecknet I finns i söksträngen på platsen 0. Därför förflyttas söksträngen åt höger (Figur 2, rad 3) och jämförelse börjar om från det sista tecknet i söksträngen.

- Om tecknet i texten som orsakade den misslyckade jämförelsen inte finns i söksträngen då förflyttas söksträng åt höger antal positioner som är lika med söksträngens längd. I Figur 2 rad 3 visas en misslyckad jämförelse där tecknet "F" från texten inte finns i söksträngen. Enligt arrayen right[] i Tabellen 3 är det -1 som motsvarar tecknet F som betyder att tecknet F inte finns i söksträngen. I detta fall flyttas söksträngen åt höger det antal positioner som är lika med söksträngens längd (rad 4 i Figur 2). Jämförelse börjar om från det sista tecknet i söksträngen.

- Om tecknet i texten som orsakar den misslyckade jämförelsen finns i söksträngen men den sista platsen i söksträngen som detta tecken finns på finns till höger om positionen där den misslyckade jämförelsen inträffade flyttas söksträngen en position åt höger (Figur 3).

1	A	R	G	E	E	S	I	Z	L	P	S	Q	F	I	L	L	N	E	S	S
2	I	L	L	N	E	S	S													
3							I	L	L	N	E	S	S							
4														I	L	L	N	E	S	S

Figur 2: Grafisk representation av Brute-force algoritmen.

1	A	R	G	E	E	S	S	S	L	P	S	Q	F	I	L	L	N	E	S	S
2	I	L	L	N	E	S	S													
3		I	L	L	N	E	S	S												

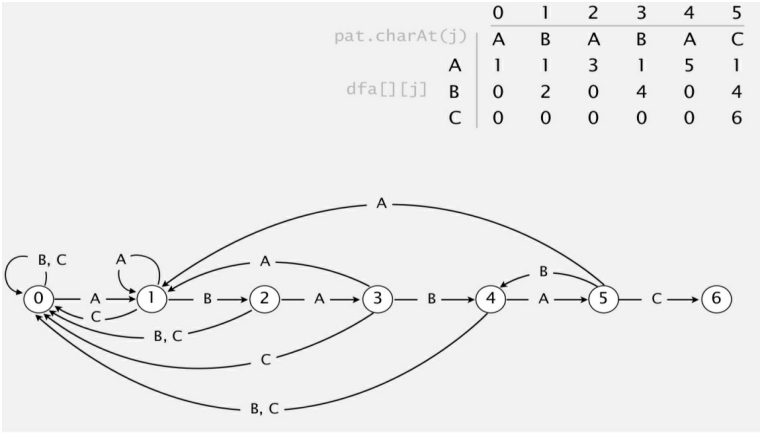
Figur 3: Grafisk representation av Boyer-Moore algoritmen.

4.3 Knuth-Morris-Pratt algoritm

- tidskomplexitet för förbehandling är $O(m * R)$ där m är söksträngslängd och R är antal tecken i alfabetet [1]

- antal jämförelser för sökningen är $O(n)$ där n är textlängd

Knuth-Morris-Pratt algoritmen har förbehandling vilket Brute-force algoritmen inte har och algoritmens idé är att när en misslyckade jämförelse händer efter minst en träff är minst ett tecken från texten redan känt [1]. Det leder till att text pekaren aldrig backar och antal jämförelser minskar.
Under förbehandlingen byggs en tvådimensionell array med heltal som kallas dfa (deterministic finite state automaton) [1] som visas i det övre högra hörnet i Figur 4. Den här arrayen anger hur mycket söksträngspekaren ska backa vid en misslyckad jämförelse. Antal rader i arrayen är 256 som representerar varje av 256 möjliga tecken vid 8-bitars teckenkodning EXTENDED ASCII, antal column är lika med söksträngslängd. Arrayens storlek kommer bli $R * m$ och den måste sparas i minnet, där R = 256 möjliga tecken och m är söksträngslängd. Arrayen byggs upp genom två for-loopar. Den yttre loopar genom hela sökströngen och skriver i matrisen vilket nästa tillstånd är vid en träff. Den inre loopen skriver i matrisen vilket är nästa tillstånd vid en misslyckad träff.
Sökningen i Knuth-Morris-Pratt algoritmen genomförs som Brute-force algoritmen från vänster till höger och koden finns i *Algoritm* 3 nedan. Figur 5 visar jämförelse som algoritmen gör för söksträngen ABABAC och texten BCBAABACAABABACAAA. Algoritmen startar med jämförelse från första tecknet i söksträngen och första tecknet i texten och söker i texten tills den hittar ett tecken som motsvarar första tecknet i söksträngen, i detta fall A. Vid match (rad 4 i Figur 5) sker övergång till tillstånd 1 och algoritmer fortsätter söka i texten och leta efter nästa tecken från söksträng som är B. När aktuellt tillstånd är 1 och tecken B läses i texten så sker övergång till tillstånd 2 (Figur 4) och algoritmen fortsätter leta efter tredje tecknet från söksträngen osv. Algoritmen är färdig när sista tillståndet har nåtts som är 6 i detta fall. Det betyder att söksträngen har hittats.



Figur 4: Grafisk representation av deterministic finite-state automaton (dfa) [1]

Text	B	C	B	A	A	B	A	C	A	A	B	A	B	A	C	A
Tillstånd	0	0	0	0	1	1	2	3	0	1	1	2	3	4	5	6
1	A	B	A	B	A	C										
2		A	B	A	B	A	C									
3			A	B	A	B	A	C								
4				A	B	A	B	A	C							
5				A	B	A	B	A	C							
6					A	B	A	B	A	C						
7					A	B	A	B	A	C						
8					A	B	A	B	A	C						
9									A	B	A	B	A	C		
10									A	B	A	B	A	C		
11										A	B	A	B	A	C	
12										A	B	A	B	A	C	
13										A	B	A	B	A	C	
14										A	B	A	B	A	C	
15										A	B	A	B	A	C	
16										A	B	A	B	A	C	

Figur 5: Grafisk representation av Knuth-Morris-Pratt algoritmen [1].

```
1  public class KMP {
2   private String pat;
3   private int [][] dfa;
4
5   public KMP(String pat) {
6       this.pat=pat;
7       int patLength = pat.length();
8       int R=256;
9       dfa=new int[R][patLength];
10      dfa[pat.charAt(0)][0]=1;
11      for (int x=0; patPointer=1; patPointer<patLength; patPointer++) {
12        for (int c=0; c<R; c++) {
13           dfa[c][patPointer]=dfa[c][x];
14        }
15        dfa[pat.charAt(patPointer)][patPointer]=patPointer+1;
16        x = dfa[pat.charAt(patPointer)][x];
17      }
18  }
19
20  public void search(String txt) {
21    int txtPointer, patPointer;
22    int txtLength = txt.length();
23    int patLength = pat.length();
24    int amountComparisons=0;
25    int amountHits = 0;
26    for(txtPointer=0, patPointer=0; txtPointer<txtLength; txtPointer++){
27      amountComparisons++;
28      patPointer = dfa[txt.charAt(txtPointer)][patPointer];
29      if (patPointer == pat.Length) {
```

```
30              amountOfHits++;
31              patternPointer = 0;
32          }
33      }
34  }
35
36  public static void main(String[] args) {
37      String txt = "ACAABABACAA";
38      String pat = "ABABAC";
39      KMP kmp = new KMP(pat);
40      kmp.search(txt);
41  }
42  }
```

Algoritm 3. Knuth-Morris-Pratt algoritm [1]

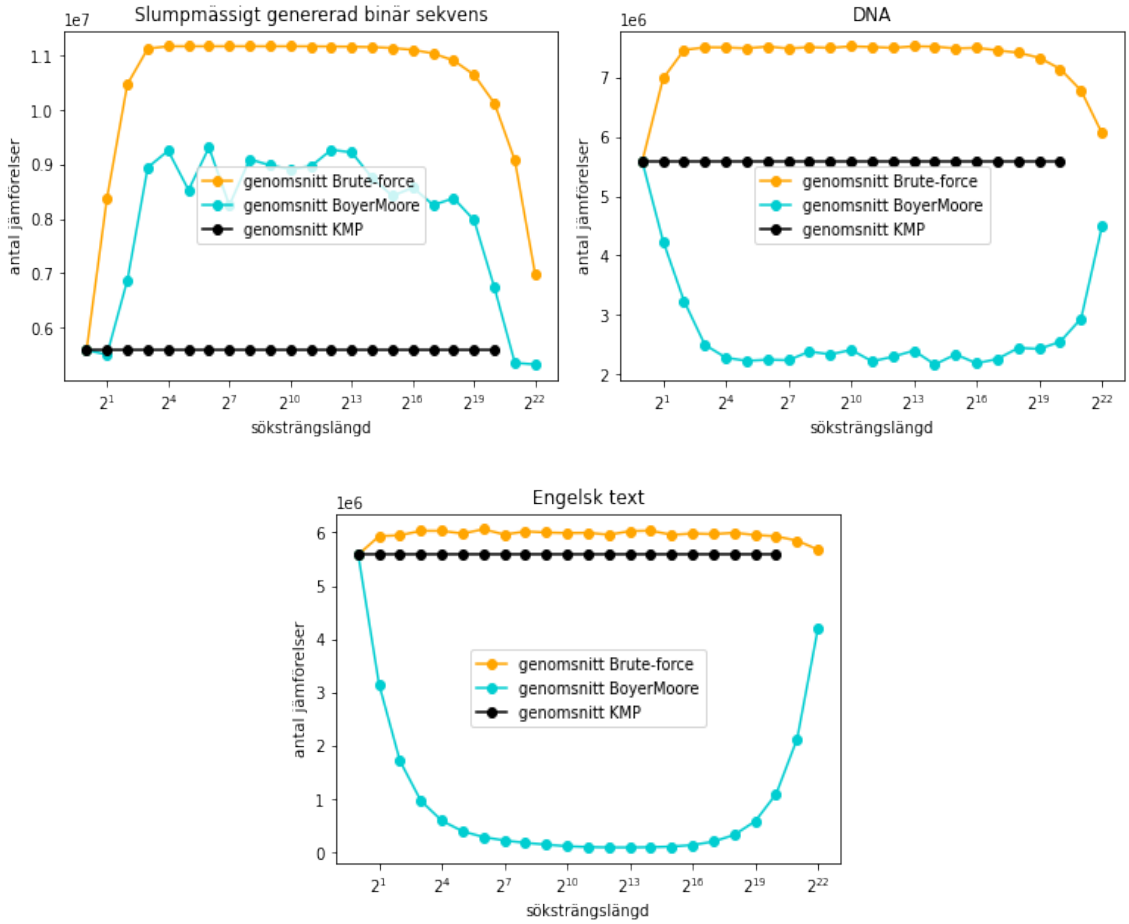
4.4 Sammanfattning

Algoritm	**Förbehandling**	**Sökning**
Brute-force	-	(m*n) i värsta fall
Boyer-Moore	m+R	(n/m) i bästa fall, $(m * n)$ i värsta fall
Knuth-Morris-Pratt	m*R	n

Tabell 4: Sammanfattning av antal jämförelser för strängssöknings algoritmer. n är textlängd, m är söksträngslängd och R är antal möjliga tecken beroende på alfabetets kodningstyp (för ASCII är R = 256).

5 Resultat

Följande avsnitt beskriver resultat av experimenten för varje typ av text: slumpmässig genererad binär sekvens, DNA sekvens och Engelsk text. Dessa resultat förklaras i analysavsnittet.



Figur 6: Antal jämförelser beroende på typ av text och söksträngslängd

.

5.1 Slumpmässig binär sekvens

Resultatet av experimentet där data representeras av slumpmässig genererad binär sekvens visar att Knuth-Morris-Pratt algoritm gör färre antal jämförelser än Boyer-Moore och Brute-force algoritmerna i nästan alla testfall oberoende av söksträngslängden (Figur 6), med undantag när söksträngslängden är ett tecken.
Antal jämförelser hos Brute-force och Boyer-Moore algoritmerna ökar snabbt när söksträngarna är korta (8 tecken) och minskar snabbt när söksträngarna är långa (524288 tecken). I mitten förändras antalet jämförelser inte som Brute-force algoritmen gör medan antalet jämförelser som Boyer-Moore algoritmen gör ändras hela tiden (Figur 6).

5.2 DNA sekvens

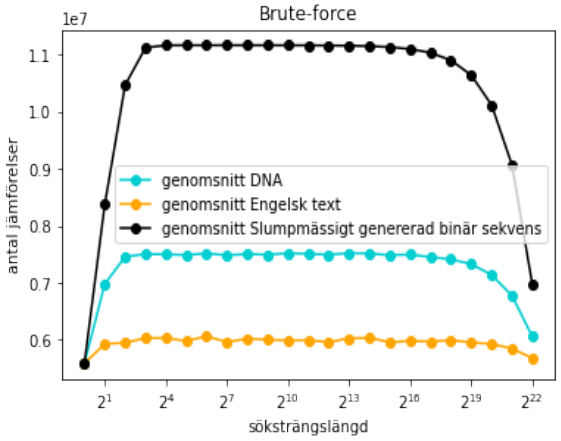
I DNA sekvens består datan av fyra bokstäver (A, C, G och T) och resultatet visar att Boyer-Moore algoritmen gör färre antal jämförelser än de andra algoritmerna (Figur 6). Antal jämförelser som Knuth-Morris-Pratt algoritmen genomför är alltid lika med textlängden där sökningen genomförs (Figur 6). Den blåa kurvan i grafen (Figur 6) visar Boyer-Moore algoritmen och den ser ut som för Engelsk text: antalet jämförelser minskar snabbt när söksträngarna är korta och antal jämförelser ökar när söksträngarna blir långa, mitten kurvan förblir ganska platt.
Brute force algoritmen gör fler jämförelser än de andra algoritmerna.

5.3 Engelsk text

Resultatet visar att Boyer-Moore algoritmen gör färre antal jämförelser än Knuth-Morris-Pratt och Brute-force algoritmerna i nästan alla testfall, utom när söksträngslängden är ett tecken då gör alla algoritmerna samma antal jämförelser (se Figur 6) och resultatet överensstämmer med resultaten som beskrivs i litteraturstudien. Antal jämförelser som Boyer-Moore algoritmen gör minskar med ökningen av söksträngslängden och minskningen är störst fram till att söksträngslängden är 8 tecken (2^3). När söksträngens storlek är mellan 8 tecken (2^3) och 65536 tecken (2^{16}) ändras antal jämförelser inte mycket och från söksträngsstorlek 65536 tecken (2^{16}) börjar antal jämförelser öka (Figur 6).
I alla testfall är antalet jämförelser för Knuth-Morris-Pratt algoritm lika med antal tecken i datan där sökningen görs och ändras inte med ändringen av söksträngslängden (Figur 6). Skillnaden mellan antal jämförelser som Knuth-Morris-Pratt algoritmen (med linjär tidskomplexitet) och Brute-force algoritmen (med kvadratisk tidskomplexitet) gör är inte betydligt stor.

6 Analys och Diskussion

6.1 Brute-force algoritm



Figur 7: Antal jämförelser för Brute-force algoritmen.

Resultatet visar att Brute-force algoritmen behöver göra fler jämförelser än de andra algoritmerna för alla tre typer av data (Figur 6) med undantag när söksträngsstorlek är ett tecken (då gör alla algoritmerna samma antal jämförelser). När söksträngens storlek ökar uppstår skillnader i antal jämförelser mellan algoritmerna, då gör Brute-force algoritmen fler jämförelser än de andra algoritmerna. Det beror på att Brute-force förflyttar söksträngen bara ett steg åt höger (*Algoritm* 1) vid både träff och misslyckad jämförelse. Boyer-Moore och Knuth-Morris-Pratt algoritmerna flyttar söksträngen några positioner åt höger. Det minsta antal jämförelser som Brute-force algoritmen gör är med data som består av Engelsk text och flest antal jämförelser på slumpmässigt genererad binär sekvens (Figur 7).

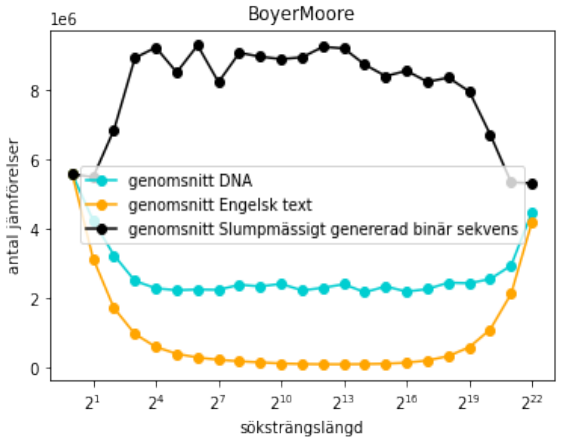
I slumpmässigt genererad binär sekvens är sannolikheten för att matcha det första tecknet $1/2$, sannolikheten att andra tecknet också matchar är $1/4$, sannolikheten för att matcha tredje tecknet är $1/8$ osv. Det betyder att den inre loopen inte gör m jämförelser, där m är söksträngslängd, utan gör $\sum_{0}^{m-1} 2^{-i}$ (där i har värde från 0 till $m-1$) som summerat blir lika med två. Darför kan antalet jämförelser för slumpmässigt genererad binär sekvens beskrivas som $(n-m) * \sum_{0}^{m-1} 2^{-i}$ där n är textlängd, m är söksträngslängd och i är från 0 till $m-1$.
Antal jämförelser för Engelsk text och DNA sekvens följer inte $(n-m) * \sum_{0}^{m-1} 2^{-i}$ och inre for-loopen för de här typerna av text gör färre jämförelser än för binär sekvens. Det kan förklaras med att i DNA sekvens och Engelsk text, som har större alfabet, är sannolikheten att inre for-loopen avbryts större. Detta leder till att antalet jämförelser kommer bli färre för DNA sekvens och Engelsk text.
När söksträngarna blir långa (från 2^{20}) kommer antal jämförelser minska för alla tre typer

av data. Det beror på att om slutet av texten är mindre än storleken av söksträngen avbryts algoritmen. Detta förklarar minskningen av antal jämförelser för de extremt stora söksträngar i slutet av kurvorna.

6.1.1 Generalisering

Brute-force algoritmen passar bra för korta söksträngar, i bästa fall när söksträngens storlek är ett tecken. I detta fall gör alla algoritmerna samma antal jämförelser men fördel med Brute-force algoritmen är att den inte gör förbehandling som kan bli tidsödande hos Boyer-Moore och mest för Knuth-Morris-Pratt algoritmen.
Datatyp när Brute-force algoritmen fungerar bäst är text med stort alfabet, som Engelsk text, då blir antal jämförelser färre än med annan typ av data från mindre alfabet. Brute-force algoritmen är mindre effektiv för slumpmässigt genererad binär sekvens. I tidigare forskning undersöktes korta söksträngar med maximala längden 100 tecken och resultatet av genomförda experiment i detta arbetet stämmer med tidigare forskningen.

6.2 Boyer-Moore algoritm



Figur 8: Antal jämförelser för Boyer-Moore algoritmen.

Kurvorna i Figur 8 har tre olika regioner:

1. korta söksträngar, från 1 tecken (2^0) till 8 tecken (2^3).
2. medelstora söksträngar, från 8 tecken (2^3) till 524288 tecken (2^{19}).
3. stora söksträngar, större än 524288 tecken (2^{19})

6.2.1 Korta strängar

När sökningen görs i DNA sekvens och Engelsk text (Figur 8) minskar antal jämförelser snabbt och det beror på alfabets storlek. Engelsk text har stor storlek på alfabetet (84 tecken) och när söksträngar är korta är det stor sannolikhet att tecken från texten som orsakar misslyckad jämförelse inte finns i strängen. I detta fall kan söksträngar förflyttas ett

antal positioner åt höger som är lika med söksträngens längd. Om en misslyckad jämförelse uppstår med första jämförelsen leder det till att antal jämförelser i bästa fall är (n/m) där n är textlängd och m är söksträngslängd. Antalet jämförelser minskar med ökningen av söksträngslängden pga ju längre söksträngen desto fler antal steg kan den förflyttas till höger i förhållande till texten.
För slumpmässig genererad binär sekvens ökar antalet jämförelser tills söksträngsstorleken blir 8 tecken (2^3). Som beskrivs ovan beror det på att alfabetets storlek för binär sekvens är kort (två tecken) och sannolikheten för en misslyckad jämförelse är mindre.

6.2.2 Medelstora strängar

I mitten av diagrammet (Figur 8) är kurvorna nästan platta (for DNA sekvens och Engelsk text). Antal jämförelser vid sökningen i binär sekvens ändras i mitten när söksträng storlek är mellan 8 (2^3) och 524288 (2^{19}) tecken. Det kan bero på att alfabetet är litet (två bokstäver) och sannolikheten att samma tecknet i texten som orsakar en misslyckad jämförelse finns i söksträngen men till höger om positionen där en misslyckad jämförelsen inträffade är högre. I detta fall förflyttas söksträng bara ett tecken åt höger och som leder till att antal jämförelser ökar (Figur 3 visar detta fallet men bara med engelsk text). Om tecknet i texten som orsakar en misslyckad jämförelse finns i söksträngen men till vänster om positionen där en misslyckad jämförelsen inträffade kan söksträngen förflyttas mer till höger och det leder till att antal jämförelser minskar.

6.2.3 Långa strängar

När söksträngarna blir stora (från 2^{19}) ökar antal jämförelser för Engelsk text och DNA sekvens. Det beror på att stora strängar finns i texten där sökningen genomförs och det förhindrar att algoritmen hoppar över tecken och Boyer-Moore algoritmen är tvungen att matcha varje enskilt tecken i den långa söksträngen. Den största söksträngen är cirka 75% av hela texten och om söksträngen var 100% av hela texten måste varje enskilt tecken jämföras. Observera att experimenten är designade på ett sätt där söksträngar måste finnas minst en gång i texten men om söksträngen inte nödvändigtvis finns i texten skulle ingen ökning av jämförelser synas i diagrammet.
Genomsnittlig tidskomplexitet för Boyer-Moore algoritmen kan bero på typ av data och textens interna struktur (slumpmässig skapad text eller riktig text som t.ex. på Engelska språket). Om datan och söksträngar genereras slumpmässigt skulle Boyer-Moore algoritmen flytta söksträngen åt höger så många steg som tecknets sannolikhet att finnas i söksträngen. På detta sätt är sannolikheten för att ett tecken matchar $1/R$ och sannolikheten för en misslyckad jämförelse är $1 - 1/R$, där R är alfabets storlek.
Om texten och söksträngen genereras oberoende av varandra och slumpmässigt (till skillnad från de nuvarande experimenten där söksträngar måste ingå i texten) kan algoritmen flytta söksträngen till höger ett stort antal steg om en misslyckad jämförelse uppstår. Algoritmen kommer att flytta söksträngen så många steg som behövs för att matcha det nuvarande tecknet från texten med tecknet i söksträngen och detta innebär att det ge-

nomsnittliga skiftet framåt kommer bli:

$$1/R * 1 + (1 - 1/R) * R$$

där den första delen (1/R*1) är sannolikheten för matchning och den andra delen ((1 − $1/R) * R$) är sannolikheten för misslyckad jämförelse. De båda delarna är multiplicerade med steg för förflyttningen, 1 för en träff och R för en misslyckad jämförelse.

$$1/R * 1 + (1 - 1/R) * R = 1/R + R - 1$$

Antal jämförelser som måste göras för att söka i hela texten är:

$$(n/(1/R + R - 1))$$

där n är textlängd och R är alfabets storlek.
Tre nästa rader visar hur antal jämförelser ändras för att söka en sträng beroende på alfabetets storlek:

$$\mathbf{R = 2:}\ n/(\frac{1}{2} + 2 - 1) = (n * \frac{2}{3}) = (n * 0.66)$$

$$\mathbf{R = 4:}\ n/(\frac{1}{4} + 4 - 1) = (n * \frac{4}{13}) = (n * 0.307)$$

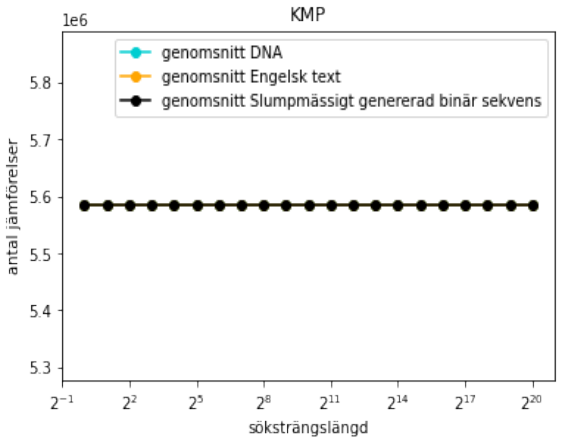
$$\mathbf{R = 84:}\ n/(\frac{1}{84} + 84 - 1) = n * (\frac{84}{6973}) = (n * 0.012)$$

Antal jämförelser som måste göras för att söka i hela texten med stora alfabet kan beskrivas som:

$$n/(\frac{1}{R} + R - 1) = n/((1 + R^2 - R)/R) = (n/R)$$

6.2.4 Generalisering

Resultatet av experiment stämmer med resultat som beskrevs i litteraturstudie som visar att Boyer-Moore algoritmen är mer effektiv för DNA sekvens och Engelsk text än för slumpmässigt skapad binär sekvens med ett undantag i källan [7]. I källan [7] Boyer-Moore algoritmen är mer effektiv även för binär text.
Experimentet visar att Boyer-Moore algoritmen är en bra algoritm när sökning genomförs i texter som består av ett stort alfabet (som Engelsk text) och för alfabet bestående av fyra tecken (DNA sekvens). Om alfabetet är litet (som slumpmässig skapad binär sekvens) är det bättre att använda Knuth-Morris-Pratt algoritmen. Om sökströngens storlek är ett tecken då är det bättre att använda Brute-force algoritmen som inte behöver förbehandling. Det beror på att när sökströngens storlek är ett tecken genomför alla algoritmerna samma antal jämförelser men fördel med Brute-force algoritmen är att den inte behöver förbehandlingen.



Figur 9: Antal jämförelser för Knuth-Morris-Pratt algoritmen.

6.3 Knuth-Morris-Pratt algoritm

Vid alla testfall med alla tre typer av av texter (binär sekvens, DNA sekvens och engelsk text) gör Knuth-Morris-Pratt algoritmen jämförelser som är lika med storleken av datamängden. Det beror på att det byggs en tillståndsmaskin och under sökningen går algoritmen genom varje tecken i texten.
Tidskomplexiteten för förbehandling för Knuth-Morris-Pratt algoritmen för att bygga en tillståndsmaskin är $(m * R)$ och tar mer tid än förbehandlingen för Boyer-Moore algoritmen som är $(m + R)$. Följaktligen ökningen av sökträngslängden leder till längre tid för förbehandlingen.

6.3.1 Generalisering

I jämförelse med andra algoritmerna är Knuth-Morris-Pratt algoritmen mer effektiv för slumpmässig genererad binär sekvens än de andra algoritmerna (Figur 6). Om det ska göras val mellan Knuth-Morris-Pratt och Boyer-Moore algoritmerna då förutom datatyp och sökträngsstorlek måste också tas hänsyn till förbehandlingen. De båda algoritmerna bygger under förbehandlingen varsin heltal array (endimensionell för Boyer-Moore och tvådimensionell för Knuth-Morris-Pratt algoritmen). Längden av den endimensionella arrayen och antal rader i den tvådimensionella arrayen betecknas båda som R i koden (i *Algoritm* 2 och *Algoritm* 3) och R=256, där 256 är möjliga tecken vid 8-bitars teckenkodning $EXTENDEDASCII$. Knuth-Morris-Pratt algoritm kräver plats i minnet som är $R * m$, där m är sökträngslängd medan Boyer-Moore algoritmen kräver $R + m$.
Resultatet av experimentet stämmer med resultatet av tidigare forskningen med ett undantag i källan [7] där Boyer-Moore algoritmen är mer effektiv även för binär text än Knuth-Morris-Pratt algoritmen.

7 Slutsatser och vidare forskning

I denna studie jämförs praktiskt tre olika strängsökningsalgoritmer: Brute-force, Boyer-Moore och Knuth-Morris-Pratt. För experiment skapades slumpmässigt söksträngar med längden från ett till 4194304 tecken. Söksträngslängden fördubblas så länge den är mindre än halva datastorleken (datastorlek är 5583968 tecken) för alla tre typer av data.

Forskningsfrågan i denna studie är:

- Hur söksträngsstorlek och datatyp påverkar strängsökningsalgoritmers effektivitet?

Genomförande av experiment visade att söksträngsstorlek och typ av data (slumpmässig skapad binär sekvens, DNA sekvens och Engelsk text) har stort betydelse för strängsökningsalgoritmers effektivitet. Det betyder att val av en algoritm beror på typ och storlek av data.

Brute-force algoritmen är effektiv för korta söksträngar, i bästa fall när söksträngens storlek är ett tecken eftersom alla algoritmerna gör samma antal jämförelser i detta fall. Fördel med Brute-force algoritmen är att den inte behöver göra förbehandling som tar tid. Boyer-Moore och Knuth-Morris-Pratt algoritmerna kräver förbehandling. Boyer-Moore algoritmen är effektiv när sökning genomförs i texter som består av ett stort alfabet (som Engelsk text) och för alfabet bestående av fyra tecken (DNA sekvens). Boyer-Moore algoritmen är effektiv när långa strängar ska sökas. Knuth-Morris-Pratt algoritmen är effektiv för slumpmässigt genererad binär sekvens.

Resultatet av experiment visade också att när söksträngarna blir stora ökar antal jämförelser för Engelsk text och DNA sekvens hos Boyer-Moore algoritmen. Det beror på att stora söksträngar finns i texten där sökningen genomförs och det förhindrar att algoritmen hoppar över flera tecken och Boyer-Moore algoritmen är tvungen att matcha varje enskild tecken i den långa söksträngen. Därför framtida experiment kan undersöka hur algoritmernas effektivitet ändras när de söker efter strängar som inte finns i texten. Annan forskning som kan göras i framtiden är att skapa en slumpmässigt genererad text för flera olika storlekar på alfabet för att undersöka hur alfabetets storlek och datans sammansättning påverkar sökalgoritmerna.

8 Referenser

[1] R. Sedgewick and K. Wayne, Algorithms, 4th ed. Boston, MA: Addison-Wesley Educational, 2011.

[2] T. Berry and S. Ravidan, "A fast string matching algorithm and experimental results", In Proceedings of the Prague Stringology Club Workshop '99. pp. 16–28.

[3] J. Tarhio, J. Holub, and E. Giaquinta, "Technology beats algorithms (in exact string matching): Technology Beats Algorithms," Softw Pract Exper, vol. 47, no. 12, pp. 1877–1885, Dec. 2017, doi: 10.1002/spe.2511.

[4] G. Navarro and M. Raffinot, Flexible Pattern Matching in Strings: Practical on-Line Search Algorithms for Texts and Biological Sequences. Cambridge University Press, 2004.

[5] B. J. Oates, Researching information systems and computing. SAGE Publications Ltd, 2006

[6] S. U. Mane and K. H. Pangu, "Disease Diagnosis Using Pattern Matching Algorithm from DNA Sequencing: a Sequential and GPGPU based Approach," in Proceedings of the International Conference on Informatics and Analytics, Pondicherry India, Aug. 2016, pp. 1–5. doi: 10.1145/2980258.2980392.

[7] P. D. Michailidis and K. G. Margaritis, "On-line string matching algorithms: survey and experimental results," International Journal of Computer Mathematics, vol. 76, no. 4, pp. 411–434, Jan. 2001, doi: 10.1080/00207160108805036.

[8] H. Lu, K. Zheng, B. Liu, X. Zhang, and Y. Liu, "A Memory-Efficient Parallel String Matching Architecture for High-Speed Intrusion Detection," IEEE J. Select. Areas Commun., vol. 24, no. 10, pp. 1793–1804, Oct. 2006, doi: 10.1109/JSAC.2006.877221.

[9] R. Vaz, V. Shah, A. Sawhney, and R. Deolekar, "Automated Big-O analysis of algorithms," in 2017 International Conference on Nascent Technologies in Engineering (ICNTE), Vashi, Navi Mumbai, India, Jan. 2017, pp. 1–6. doi: 10.1109/ICNTE.2017.7947882.

[10] M. Parker and C. Lewis, "Why is big-O analysis hard?," in Proceedings of the 13th Koli Calling International Conference on Computing Education Research - Koli Calling '13, Koli, Finland, 2013, pp. 201–202. doi: 10.1145/2526968.2526996.

[11] D. E. Knuth, "Big Omicron and big Omega and big Theta". SIGACT News 8, 2 (April-June 1976), pp.18–24.

[12] R.A. Baeza-Yates, "Algorithms for String Searching: A Survey", SIGIR Forum 23, 3–4 (Spring 1989), pp.34–58.

[13] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," Commun. ACM,

vol. 20, no. 10, pp. 762–772, Oct. 1977, doi: 10.1145/359842.359859.

[14] D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt, "Fast Pattern Matching in Strings," SIAM J. Comput., vol. 6, no. 2, pp. 323–350, Jun. 1977, doi: 10.1137/0206024.

[15] S.S. Sheik, S.K. Aggarwal, A. Poddar, B. Sathiyabhama, N. Balakrishnan, K. Sekar, "Analysis of string-searching algorithms on biological sequence databases", Current Science 89, no. 2 (July 25, 2005), pp.368–374.

[16] D. Gusfield, Algorithms on strings, trees, and sequences: Computer science and computational biology. Cambridge Univ. Press, 1997.

[17] Princeton.edu. [Online]. Available: https://introcs.cs.princeton.edu/java/data/chromosome4.txt. [Accessed: 12-May-2021].

[18] Princeton.edu. [Online].
Available: https://introcs.cs.princeton.edu/java/data/leipzig/leipzig1m.txt. [Accessed: 12-May-2021].

[19] "System (Java Platform SE 7)," Oracle.com, 24-Jun-2020. [Online].
Available: https://docs.oracle.com/javase/7/docs/api/java/lang/System. html. [Accessed: 09-May-2021].

[20] "Internet: most common languages online 2020," Statista.com. [Online]. Available: https://www.statista.com/statistics/262946/share-of-the-most-common-languages-on-the-internet/. [Accessed: 09-May-2021].

[21] "What are the top 200 most spoken languages?," Ethnologue.com, 03-Oct-2018. [Online]. Available: https://www.ethnologue.com/guides/ethnologue200. [Accessed: 09-May-2021].