Hur kan jag göra datamaskiner snabbare?

Baserat på verkliga händelser (högst 90% fabrikation)

Alternativt: Snabba data strukturer för medicin och bioinformatik[*]

Saska Dönges

[*]Med material stulet från Travis Gagies DCC 2023 inledningsanförande

AGENDA

1. Vem är jag?
2. Berättelsen
3. Motivation
4. Resultat

JAG

Doktorand i “Compressed Data Structures” gruppen, som är en del av supergruppen “Algorithmic Bioinformatics”.

Borde disputera inom $\sim$ 1.5 år.

Efter gymnasiet och milin studerade jag utomlands i några år tills jag burnouttade och flyttade tillbaka till Finland.

Jobbade några år “inom industrin” tills jag fick nog.

Startade på uni hösten 2015 $\rightarrow$ magister på våren 2021.

BERÄTTELSEN

Jag återkommer till medicin, bioinformatik och mer detaljerad information om forskningsresultat om jag har tid.

Men först, en berättelse om min forskning, som kanske skulle kunna vara sann.

DATAMASKINER ÄR SNABBA

En modern dator kan göra miljontals meningsfulla beräkningar per sekund.

Men datorer är dumma. De gör exakt vad som bes av dem.

Även om det skulle kunna finnas bättre sätt att göra saker.

Gör en svår sak enklare för datorn $\Leftrightarrow$ Datorn kan beräkna saken snabbare.

MULTIPLIKATION

Det är (inte speciellt) svårt att beräkna $70 \cdot 300$.

Det är enklare att räkna $7 \cdot 3 \cdot 10 \cdot 100$. “21 och tre nollor” $= 21000$.

Skulle det här vara något som vi skulle kunna hjälpa datorer med?

MULTIPLIKATION

Det är (inte speciellt) svårt att beräkna $70 \cdot 300$.

Det är enklare att räkna $7 \cdot 3 \cdot 10 \cdot 100$. “21 och tre nollor” $= 21000$.

Skulle det här vara något som vi skulle kunna hjälpa datorer med?

Nej. Tyvärr är datorer bra på att multiplicera, och bryr sig inte egentligen om vilka tal som används.

MULTIPLIKATION

Det är (inte speciellt) svårt att beräkna $70 \cdot 300$.

Det är enklare att räkna $7 \cdot 3 \cdot 10 \cdot 100$. “21 och tre nollor” $= 21000$.

Skulle det här vara något som vi skulle kunna hjälpa datorer med?

Nej. Tyvärr är datorer bra på att multiplicera, och bryr sig inte egentligen om vilka tal som används.

Men kanske division...

DIVISION

Det är (inte speciellt) svårt att beräkna $1000/300$.

Det är enklare att ta $\frac{1000/100}{3}$. “Ta bort 2 nollor och dividera med 3” $= 10/3 = 3\frac{1}{3}$.

Skulle det här vara något som vi skulle kunna hjälpa datorer med?

DIVISION

Det är (inte speciellt) svårt att beräkna $1000/300$.

Det är enklare att ta $\frac{1000/100}{3}$. “Ta bort 2 nollor och dividera med 3” $= 10/3 = 3\frac{1}{3}$.

Skulle det här vara något som vi skulle kunna hjälpa datorer med?

Jo! Fast inte så som jag gjorde ovan.

DIVISION

Det visar sig att datorer inte är speciellt bra på att dividera. Det skulle vara bättre att multiplicera i stället.

Så i stället för att beräkna $1000/300$ kan vi beräkna $1000 \cdot 0.0033333\ldots$.

Eller om vi arbetar med 32-bitars heltal kan vi beräkna $1000 \cdot 458129845 \gg 37$ I stället.

Alltså vi sparar resultatet av $1000 \cdot 458129845$ som ett 64-bitars heltal och skiftar sedan resultatet till höger med 37 bitar.

DIVISION

$$
\begin{array}{rl}
 & 00000000000000000000000000000000000000000000000010011100010000 \\
/ & 00000000000000000000000000000000000000000000000000100101100 \\
\hline
= & 00000000000000000000000000000000000000000000000000000100001
\end{array}
$$

$$
\begin{array}{rl}
 & 00000000000000000000000000000000000000000000000010011100010000 \\
\cdot & 00000000000000000000000000000000000011011010011101000000110110101 \\
\hline
= & 00000000000000000000100001010101010101010101010101011001010000
\end{array}
$$

Tyvärr gör moderna kompilatorer det här automatiskt. (Det var inte jag som listade ut lösningen på förra sidan.)

NÅ MEN STRÄNGÖKNING DÅ?

Strängsökning går ut på att hitta förekomster för en söksträng P med längden m ur en (ofta lång) text T med längden n.

Alternativt bara ta reda på om P förekommer i T, eller hur många gånger P förekommer i T.

NAIV ALGORITM FÖR STRÄNGSÖKING

En så kallad “brute-force” algoritm:

```
def str_cont(T, P):
    for i in range(len(T) - len(P)):
        for j in range(len(P)):
            if (T[i + j] != P[j]):
                break
        else:
            return True
```

VAD GÖR ALGORITMEN?

Sökningen sker genom att jämföra varje position inom T med P

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaab b matchade inte så vi hoppar till nästa position.
 aaaaab b matchade inte.
  aaaaab b matchade inte.
   aaaaab b matchade inte.
    aaaaab b matchade inte.
```

Sökningen tar i värsta fall ungefär $c \cdot n \cdot m$ sekunder (där c är en liten konstant. $\sim 1/10000000$ i det här fallet.)

HUR SNABB ÄR "BRUTE FORCE" ALGORITMEN?

```
T = "a" * 100000
P = "a" * 99 + "b"
```

T är hundratusen "a"n och *P* är 99 "a" med ett "b" på slutet.

```
%%timeit
str_cont(T, P)
```

```
1.45 s ± 3.12 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

VAD KAN VI GÖRA I STÄLLET FÖR "BRUTE-FORCE"?

I exemplet märkte ni eventuellt att vi använder "massor" med resurser för att jämföra a bokstäver som alltid matchar.
Då b bokstaven inte matchar "vet" vi ju att om fem stycken a matchade innan så matchar åtminstone fyra stycken i nästa position.
Vi kan i hoppa över alla de gråa a bokstäverna som vi vet att kommer att matcha.

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaab b matchade inte så vi hoppar till nästa position.
 aaaaab b matchade inte.
  aaaaab b matchade inte.
   aaaaab b matchade inte.
    aaaaab b matchade inte.
```

TYVÄRR(?) HAR SMARTA MÄNNISKOR REDAN TIDIGARE MÄRKT DETTA.

1970 hittade James Morris och Donald Knuth på en bättre lösning oberoende av varandra. År 1977 publicerade Knuth och Morris tilsammans med Vaughan Pratt ett papper där en effektiv version av vad som nu kallas för KMP algoritmen.

Denna, eller någon nyare (och ännu bättre) lösning är redan för det mesta implementerad i standardbibliotek av programmeringsspråk.

```
%%timeit
hittades = P in T

403 µs ± 1.86 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

MEN VAD OM VI KÄNNER TILL T PÅ FÖRHAND?

Hur ofta får vi egentligen bara ett T och P ur det blå för att genomföra en sökning?

I en analog värld där T skulle kunna vara en bok, skulle vi bläddra bak i boken och hoppas att där finns ett index där vi kan slå upp P.

Kan vi bygga ett index för T så att vi för alla framtida sökningar snabbare kan ta reda på om P förekommer i T?

MEN VAD OM VI KÄNNER TILL T PÅ FÖRHAND?

Hur ofta får vi egentligen bara ett T och P ur det blå för att genomföra en sökning?

I en analog värld där T skulle kunna vara en bok, skulle vi bläddra bak i boken och hoppas att där finns ett index där vi kan slå upp P.

Kan vi bygga ett index för T så att vi för alla framtida sökningar snabbare kan ta reda på om P förekommer i T?

Det är klart vi kan! Annars skulle berättelsen ha ett ganska snopet slut.

BURROWS-WHEELER!

Burrows-Wheeler transformen B av T är en permutation av T där bokstaven i position k inom B är bokstaven före den k:nde slutdelen, då alla slutdelar av T sorteras.

Lite jobbig definition.

Om $T =$ “abrakadabra ” så får vi

$B =$ “ard kraaaabb”.

B	slutdelarna
a	
r	a
d	abra
	abrakadabra
k	adabra
r	akadabra
a	bra
a	brakadabra
a	dabra
a	kadabra
b	ra
b	rakadabra

VAD GÖR VI MED B?

Det visar sig att B ofta är lätt att komprimera. Till exempel genom att i stället för att spara “aaaa” bara spara a^4.

Då skulle vi kunna spara B som $a^1r^1d^1\ {}^1k^1r^1a^4b^2$. Vilket kanske inte verkar så fint.

Men för praktiska data sparar detta massor med utrymme.

MEN VARFÖR VILL VI ENS HA B

Om vi har B och vet hur många av vilka bokstäver som finns i B, kan vi mycket snabbt räkna ut om P förekommer i T för ett godtyckligt P. Vi kan även enkelt ta reda på hur många gånger P förekommer i T.

Med “normal” strängsökning kan vi (långsamt) räkna hur många gånger “Einstein“ förekommer i `einstein.en.txt`[†]

```
grep -o 'Einstein' einstein.en.txt | wc -l
2475825
real        0m1.035s
user        0m1.004s
sys         0m0.053s
```

[†] `http://pizzachili.dcc.uchile.cl/repcorpus.html`

HUR SNABBA ÄR INDEX STRUKTURER FÖR STRÄNGSÖKNING?

Med ett Burrows-wheeler index kan vi räkna förekomsterna aningen snabbare:

```
time ./counting_exp einstein.en
pattern   count    time
Einstein  2475825  34649
Mean query time: 34649ns
 with 0.018994 bits per symbol.

real    0m0.006s
user    0m0.000s
sys     0m0.006s
```

ÄNNU SNABBARE!?

Men vad hände med att göra datorer snabbare?

Nu har jag bara beskrivit helt nya sätt att göra mer eller mindre samma sak?

Det visar sig att strukturen jag demonstrerade är “teoretiskt optimal” för vad den är.

Den tar så lite rum som möjligt och räknar förekomsterna i högst $c \cdot m$ sekunder (för någon konstant c).

HUR GÖR JAG DATORER SNABBARE?

Den teoretiskt optimala lösningen är inte optimal för datorn. Datastrukturen är byggd på ett sätt som gör att datorn konstant måste söka saker från olika delar av minnet för att göra de krävda beräkningarna.

(Som att bläddra fram och tillbaka i en bok om en text, bilder och tabeller som har att göra med samma sak är på helt olika sidor.)

Vi har designat en ny version av samma data struktur som går att använda för att beräkna förekomsterna i $c \cdot b \cdot m$ sekunder, där b kan vara till exempel 32. Om c för vår struktur 32 gånger mindre än tidigare versioner, är vår datastruktur snabbare.

HUR SNABB ÄR DEN DÅ?

```
time ./count_matches -c einstein.en.runs einstein.en.patterns
looking for patterns from einstein.en.patterns in einstein.en.runs
Pattern   count    time
Einstein  2475825  8305
Mean query time: 8305 / 1 = 8305ns
 with 0.0235753 bits per symbol
real    0m0.003s
user    0m0.003s
sys     0m0.000s
```

Det visar sig att *c* tydligen blir över hundra gånger mindre.

JAG FÅR (HOPPELIGEN) ÅKA TILL BARCELÅNA!

Vi har skrivit ett papper som jag hoppeligen får presentera i Juli.

DNA SEKVENSERING

Då en människas genom sekvenseras, görs det i allmänhet med hjälp av ett referensgenom.

En sekvenseringsmaskin läser in små snuttar av en ny människas dna.

Rätt plats för snuttarna söks genom att jämföra till referensen.

referens	G	A	T	A	C	A	T
snutt 1	G	A	T	A			
snutt 2		A	T	A	C		
snutt 3				A	C	A	T

DNA SEKVENSERING

Fungerar ofta ganska bra.

Men om den nya individen har små mutationer som inte kommer överens med referensen kan det uppstå problem.

referens	G	A	T	T	A	C	A	T
snutt 1	G	A	T	-	A			
snutt 2		A	-	T	A	G		
snutt 3					A	G	A	T
resultat	G	A	T	-	A	C	A	T

I problemfall fyller man bara i med referensgenomen.

REFERENS BIAS

To the scientists' puzzlement, however, the boy's sequence showed no sign of the mutation in the gene known to cause Baratela Scott, called XYLT1. Nor did the DNA of the next boy with the disorder, or the next. As they tried to compare the boys' DNA sequences to the reference genome, it was like trying to check a spelling in a Webster's from which a prankster had torn handfuls of pages. Many pieces of the boys' genomes, called short reads, "weren't in the reference genome at all," . . . There was no way to check them for disease-causing misspellings.

— Sharon Begley, *Stat News*, March 11th, 2019

Referens bias

This bias limits the kind of genetic variation that can be detected, leaving some patients without diagnoses and potentially without proper treatment. What is more, people who share less ancestry with the man from Buffalo will probably benefit less from the incoming era of precision medicine, which promises to tailor healthcare to individuals.

[O]ur understanding of diversity within populations of European descent is now so good that we can start to use it for precision medicine. But for other populations, "We do not have the same kind of data . . . [This] is going to increase healthcare disparities above and beyond what they are today." . . . [A] huge new project is offering a different solution with the aim to represent global diversity: a human pangenome.

— Ida Emilie Steinmark, *Guardian*, January 29th, 2023

REFERENS BIAS

[T]he project is not just about sequencing more diverse data. "We need to come up with a better data structure to encode that information," . . . That data structure is called a genome graph. In contrast to the current reference, which is just a long string of letters, the genome graph shows variation between genomes as detours on an otherwise shared path. That will enable researchers and doctors to map short reads to the version of the path that best fits their sample.

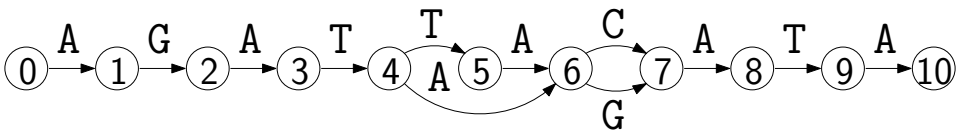
— Ida Emilie Steinmark, *Guardian*, January 29th, 2023

PANGENOM GRAF

Flera referensgenom:

- `GATTACAT`
- `AGATACAT`
- `GATACAT`
- `GATTAGAT`
- `GATTAGATA`

Pangenom graf utgående från referenserna:

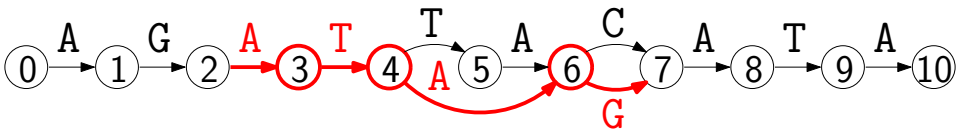


PANGENOM GRAF

Referensgenom:

- GA3TTACAT
- AGATACAT
- GATACAT
- GATTAGAT
- GATTAGATA

Ser ut som om “ATAG” skulle vara en känd delsekvens

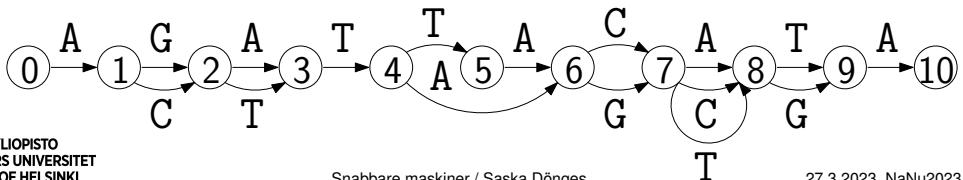


PANGENOM GRAF

Änne flera referensgenom:

- GATTACAT
- AGATACAT
- GATACAT
- GATTAGAT
- GATTAGATA
- CATTACAT
- GTTAGAT
- GATTCCATA
- GATTACAGA

Ännu svårare att veta vad som är riktigt och vad som inte är det



PANGENOM LÖSNING

Lösningen är att inte använda grafen för att hitta var en snutt hör till, utan att helt enkelt leta rätt på positionen genom att söka i alla referensgenom.

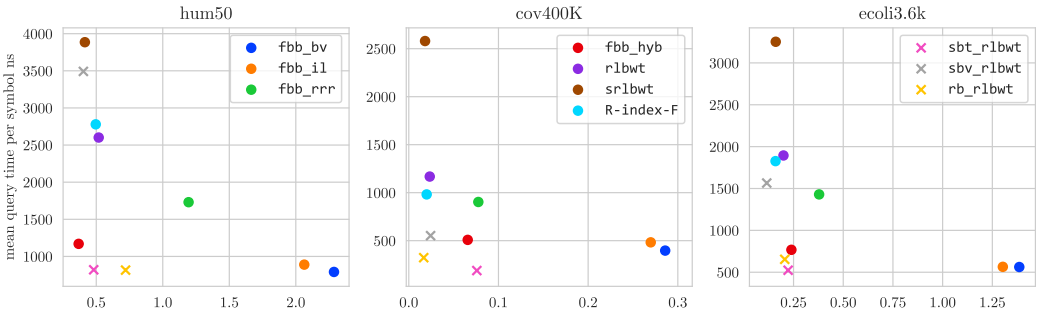
Problemet är att ett genom är hyfsat stort $\sim$ 700 Megabit (lätt komprimerat).

Om vi vill söka i 100000 referensgenom[‡], tar lätt komprimerade referenserna 70 terabyte utrymme.

Ett effektivt komprimerat index kanske bara tar $\sim$ 1 terabyte, vilket fortfarande är ganska mycket, men ryms i minnet av superdatorer.

[‡] `https://www.genomicsengland.co.uk/initiatives/100000-genomes-project`

VÅR IMPLEMENTATION ÄR BRA



Våra index indikeras med “x”. De kräver mindre utrymme och är lika snabba som de snabbaste konkurrenterna.