Avsluttende eksamen i TDT4125 Algoritmekonstruksjon, videregående kurs (LF)

Eksamensdato	15. mai 2009
Eksamenstid	0900–1300
Sensurdato	5. juni
Språk/målform	Bokmål
Kontakt under eksamen	Magnus Lie Hetland (tlf. 91851949)
Tillatte hjelpemidler	Alle trykte/håndskrevne; bestemt, enkel kalkulator

Vennligst les hele oppgavesettet før du begynner, disponer tiden og forbered evt. spørsmål til faglærer kommer til eksamenslokalet. Gjør antagelser der det er nødvendig. Skriv kort og konsist. Lange forklaringer og utledninger som ikke direkte besvarer oppgaven tillegges liten eller ingen vekt.

Oppgave 1

Vi skal lage en visualiseringsløsning med tilsvarende funksjonalitet som Google Earth for bruk i en flysimulator. Til enhver tid observerer vi et utsnitt av jordkloden, og vi ønsker å lagre terrengdataene vi trenger for å i en forhåndsallokert cache-struktur. Denne strukturen har en fastsatt minnekapasitet C og skal ha metodene SET og GET, hvor SET plasserer data i cachen og GET henter disse ut igjen. Dersom dataene ikke finnes i cachen, returnerer GET en null-verdi.

For denne cache-strukturen kan vi velge mellom mellom utkastelsesstrategiene *least recently used* (LRU) – utkastelse basert på det elementet det er lengst tid siden ble aksessert – eller *least frequently used* (LFU), implementert med utkastelse basert på det totalt laveste antallet forespørsler.

a. Velg en av disse strategiene, og forklar kort hvorfor denne sannsynligvis vil fungere best for vår applikasjon.

Kandidaten bør ha fått med seg:

- LRU
- Dataene har god lokalitet
- $O(1)$

b. Ta utgangspunkt i cache-strategien du valte i oppgave **a**. Angi hvilke datastrukturer vi trenger, og skisser implementasjonen for funksjonen GET.

Hvis LRU:

Datastrukturene vi trenger er en hash-tabell og en dobbel-lenket liste. Studenten bør også beskrive datastrukturen vi lagrer i hashen, basert på nøkkelen.

hash(*key*) → (*listptr*, *value*):
if key not in hash
  return null

flytt *listptr* fra hvor den måtte befinne seg i listen til starten av listen
return value

(Ikke LF for LFU.)

«*Scan resistance*» kan defineres som motstandsdyktighet mot en serie av forespørsler til cachen for forskjellige datanøkler man bruker bare en gang, og med flere forespørsler enn cachens kapasitet C.

c. Beskriv en utkastelsesstrategi som er *scan resistant*, og med samme forventede kjøretid som i oppgave **a**.

Dette er et forslag, flere løsninger kan oppnå samme resultat. Men for å få $O(1)$ oppslag, er det nødvendig med en eller annen LRU-kombinasjon.

Lag to LRU-cacher, med kapasitet $C/2$, LRU1st og LRUAfter

Den første er for first-time hits, den andre for alle etterfølgende hits.

Det viktige med en implementasjon er at i GET aksesseres LRU1st først og deretter LRUAfter:

```
GET(KEY):
if LRU1st.get(KEY)
   LRUAfter.set(KEY, LRU1st.get(KEY))
   LRU1st.remove(KEY)
   return LRUAfter.get(KEY)
else if LRUAfter.get(KEY)
   return LRUAfter.get(KEY)
return null
```

```
SET(KEY, payload):
LRU1st.set(KEY, payload)
```

Oppgave 2

Du har blitt kontaktet av en biolog som studerer nettverk av proteiner. Biologen er spesielt interessert i å finne store grupper av proteiner hvor alle proteinene interagerer med hverandre. «Aha! Klikker,» tenker du, og ber raskt biologen om å sende deg en beskrivelse av nettverket som du oversetter til en graf G med proteiner som noder V og interaksjoner som kanter E. Siden du nettopp har fått gode resultater ved å bruke simulert størkning (*simulated annealing*) til å løse TSP, ønsker du også å gjenbruke programmet ditt for simulert størkning for å finne den maksimale klikken i G. Det eneste du trenger å gjøre er å definere input, i form av løsningsrepresentasjon, kostnadsfunksjon og naboskapsrelasjon.

a. Skisser en egnet løsningsrepresentasjon S, gi et eksempel på en egnet starttilstand S_0, og definer en kostnadsfunksjon $c(S)$ og naboskapsrelasjon $S \sim S'$.

$S = s_1 \ldots s_{|V|}$, $s_i \in \{0,1\}$ ($s_i = 1$ betyr at node i er med i løsningen)

$S_0 = 10000$ (for graf med 5 noder)

$c(S) = |V| - \sum s_i$ (gitt gjenbruk av kode fra TSP løsningen)

$S \sim S'$ hvis $S = S' \pm$ en node og S' er en klikk.

Etter å ha presentert din løsning for biologen, viser det seg at han er litt misfornøyd med resultatet. Det han egentlig var på jakt etter var proteiner hvor de aller fleste, men ikke nødvendigvis alle, interagerer med hverandre.

b. Definer en ny kostnadsfunksjon $c(S)$ som kan løse dette problemet. Kan du gjenbruke den eksisterende naboskapsrelasjonen (fra **a**)? Hvis ikke, definer en ny.

Eks: $c(S) = |V| - \sum \{e_i, j: e_i, j \in E \wedge s_i = 1 \wedge s_j = 1\} / \sum s_i$ (antall kanter i løsningen / antall noder i løsningen; forutsetter minimering)

Nei: $S \sim S'$ hvis $S = S' \pm$ en node.

Oppgave 3

Anta følgende sorterte sekvens med 16 tall:

29867
29869
29880
30119
30265
30473
30476
30477
30478
30481
30503
30514
30672
30682
31033
31034

Hvert tall representeres med 32 bit.

a. Hvor mange byte tar lista over?

b. Bruk deltakoding og komprimer sekvensen med vByte-koding. Hvor mange byte behøver den komprimerte sekvensen? Vis de viktigste trinnene i utregningen.

a. Komprimer sekvensen med PFOR-DELTA-koding. Anta en blokkstørrelse på 16 tall og bruk 2 elementer i unntakslista. Hvor mange bit trenger hvert element i *code section*? Hvor mange byte er det totalt behov for inkludert unntakslista? Ikke regn med noen *header*. Vis de viktigste trinnene i utregningen.

Verdi	Deltakodet	Code section	Entry point
29867	29867	13	0
29869	2	2	
29880	11	11	
30119	239	239	
30265	146	146	Unntaksliste
30473	208	208	29867

30476	3	3	351
30477	1	1	
30478	1	1	
30481	3	3	
30503	22	22	
30514	11	11	
30672	158	158	
30682	10	10	
31033	351	1	
31034	1	1	

Fjerner de 2 største tallene (29867 og 351). Det største tallet blir da 239. Trenger da 8 bit i code section.

Code section + Entry point + Unntaksliste =
8bit*16/8bit + 1*1byte + 2*4byte = 25 byte

b. Hvorfor vil PFOR-DELTA være mye raskere enn vByte?

Oppgave 4

Du skal finne en approksimasjonsalgoritme for en variant av SET-COVER-problemet der man har lagt til ett krav til probleminstansene: Hvert element i mengden X forekommer i maksimalt k av mengdene i $\mathcal{F}$, for en gitt k.

Hentet fra http://www.cs.uiuc.edu/class/sp09/cs598csc/Lectures/lecture_3.pdf

Svarene nedenfor er grundigere enn det som forventes av studentene.

a. Hvordan vil du løse problemet for $k = 2$? Hva blir kjøretid og *approximation ratio* for algoritmen? Begrunn svaret.

(Merk: Her har noen forslått å bruke den vanlige SET-COVER-approksimeringen, med ratio $H(2)$, siden største mengde har størrelse 2. Det er ikke riktig – selv om hvert element bare kan forekomme i to mengder kan mengdestørrelsen like fullt være større enn dette.)

Hvis vi antar at alle forekomstene er nøyaktig 2 har vi egentlig en instans av VERTEX-COVER (hver mengde S i $\mathcal{F}$ er kantene som dekkes av en gitt node, og hver kant er med i nøyaktig to av disse mengdene) og vi kan bruke APPROX-VERTEX-COVER. Vi må vise at løsningen fortsatt er riktig om $k = 2$ gir *maks*-frekvensen, altså om vi også kan ha frekvenser på 1 («kanter» med én «endenode»).

I originalen har vi følgende utregning:

$$|C| = 2|A| \leq 2|C^*|,$$

som gir oss en bound på 2. Det eneste som endrer seg nå er at vi i stedet får følgende:

$$|C| \leq 2|A| \leq 2|C^*|,$$

siden hver kant i $|A|$ nå resulterer i *maksimalt* 2 noder i C.

b. Hvordan vil du løse problemet generelt? Hva blir kjøretid og *approximation ratio* for algoritmen?

Vi kan fortsatt bruke samme tilnærming (dette er egentlig VERTEX-COVER i en hypergraf). I hver iterasjon:

> Velg et element x i X (tilsvarer en kant)
>
> Legg til i C alle S som inneholder x (tilsvarer nabonoder, maks k)
>
> Fjern fra X alle elementer som dekkes av disse S-ene.

Kjøretiden er fortsatt $O(V + E)$. Approximation ratio får vi ved samme logikk som før: Den optimale løsningen C* må inneholde minst ett «endepunkt» for hver av «kantene». Ingen av «kantene» i løsningen deler «endepunkter». Så ingen av «kantene» i A dekkes av den samme «noden» i C, så vi har fortsatt at

$$|C^*| \geq |A|.$$

Hver «kant» som velges finnes ingen av de opptil k endepunktene med i C fra før. Med andre ord har vi at

$$|C| \leq k|A|.$$

Dette gir oss

$$|C| \leq k|A| \leq k|C^*|,$$

som altså gir oss en bound på k.

Oppgave 5

Anta at du skal finne maksimum av n elementer med en enkel, trådbasert splitt-og-hersk-algoritme (med en todeling av datasettet, og to rekursive kall som kan kjøres i parallell).

a. I hvilken grad vil denne algoritmen kunne dra nytte av en økning i antall prosessorer? Begrunn svaret.

Vi bruker her formalismene fra Cilk. Parallellitetsgraden (potensiell speedup) er T_1/T_∞, der T_1 er arbeidet (antall noder i kallgrafen) og T_∞ er lengden på den kritiske stien. I dette tilfellet har vi $T_1 = O(n)$ og $T_\infty = O(\log n)$, som gir en parallellitetsgrad på $P^* = O(n/\log n)$. En grådig scheduler vil altså oppnå lineær speedup for $O(n/\log n)$ prosessorer. Man kan trygt konkludere med at algoritmen i stor grad kan nyttiggjøre seg en økning i antall prosessorer.

b. Vis hvordan du kan finne maksimum av av n elementer i $O(1)$ parallell tid med n^{1+c} prosessorer, der c er en vilkårlig (gitt) positiv konstant. Hvilken PRAM-modell trenger du?

Dette er oppgave 2.36 fra JaJa. For $c \geq 1$ kan man bruke algoritme 2.8 direkte. Problemet er altså å finne en løsning for $c < 1$. Ved å partisjonere elementene i $n^{½}$ grupper vil man med $O(n)$ prosessorer kunne finne maksimum i hver av dem i $O(1)$ tid, og de kan også kombineres i $O(1)$ tid. (Med andre ord kunne vi egentlig ha satt $c = 1$).

Her er det et «smutthull» som oppgaven ikke tok høyde for: I følge pensum kan man bruke såkalt *combining* CRCW med maksimum som kombinasjons-funksjon. Dermed kan man helt direkte løse problemet med n prosessorer. Selv

om dette ikke er den tenkte løsningen er den helt korrekt, gitt premissene i oppgaven, og får dermed full uttelling. (Samme løsning vil, naturligvis, også gi full uttelling på 5c.)

Oppgave 6

Oppgaven går ut på å lage en algoritme for å tilpasse støyfylte tidsserier med glatte, konvekse funksjoner. En tidsserie er gitt ved:

$$X = x_1, x_2, \ldots, x_n,$$

der x_i er heltall i verdiområdet 0 til 100 ($0 \le x_i \le M = 100$).

Tilsvarende kan den glatte og konvekse funksjonen som din algoritme skal beregne ut fra X beskrives som:

$$Y = y_1, y_2, \ldots, y_n,$$

der y_i er heltall i verdiområdet 0 til 100 ($0 \le y_i \le M = 100$).

Oppgaven går ut på å beskrive en algoritme som finner den Y som «krummer oppover» og tilpasser X best mulig. Vi vil måle tilpasningen mellom Y og X med total kvadratfeil:

(A) $E = \sum_{i=1\ldots n} (y_i - x_i)^2$

Kravet om at Y skal være konveks (dvs. «krumme oppover») betyr at den skal ha ikke-negativ andre derivert. Numerisk gir dette følgende krav:

(B) $2 \cdot y_i \le y_{i-1} + y_{i+1}$ for $i=2, \ldots, n-1$

a. Beskriv en best mulig algoritme som ut fra X finner den tidsserien Y som tilfredsstiller krav B ovenfor og har minst mulig kvadratfeil gitt ved A. Angi også hvilken kjøretid og plassbehov algoritmen din har som funksjon av n og M.

for i, j: $0 \le i \le M$ **and** $0 \le j \le M$
  $E_2(i, j) = (i-x_1)^2 + (j-x_2)^2$

for k: $3 \ldots n$
  for i, j: $0 \le i \le M$ **and** $0 \le j \le M$
    $E_k(i, j) = (i-x_{k-1})^2 + (j-x_k)^2 + \min_{q \ge 2 \cdot i - j} E_{k-1}(q, i)$
    $P_k(i, j) = q_{\min}$ (hvor $q_{\min}$ er den q som ga min-verdien i linjen over)

Finn i, j som har minimal $E_n(i, j)$
$y_n = j$
$y_{n-1} = i$

for k: $n-2 \ldots 1$
  $q = P_k(i, j)$
  $y_k = q$
  $j = i$
  $i = q$

Kjøretid er $O(n \cdot M^2)$ da min-beregningen kan gjøres i $O(1)$ ved å utnytte forrige (i, j)-beregning. Plassbehov også $O(n \cdot M^2)$. Problemet kan også løses med approksimative algoritmer og muligens en veldig direkte lineær approksimasjon med utgangspunkt i gjennomgått problem rundt å approkismere tilsvarende tidsserier med monotont voksende funksjoner.