T A R T U Ü L I K O O L

MATEMAATIKA-INFORMAATIKATEADUSKOND

Arvutiteaduse instituut

Tarkvarasüsteemide õppetool

Informaatika eriala

Jüri Reimand

Kiire hulgaaritmeetika ja rakendused andmekaeves

Semestritöö

Juhendaja: Jaak Vilo, PhD

Autor: ..... “.....” mai 2004

Juhendaja: ..... “.....” mai 2004

Õppetooli juhataja: ..... “.....” 2004

TARTU 2004

Sisukord

1 Sissejuhatus

Tänapäeva ühiskonnale on iseloomulikuks informatsiooni üleküllus. Kättesaadavate andmehulkade maht kasvab kiiresti ning üha suurenev osa ajast tuleb kulutada infootsinguks ja relevantsete infoosakeste eraldamiseks mürast. Sageli ei ole infootsing inimesele jõukohane ning abiks tuleb võtta automatiseeritud vahendid.

Lisaks otseselt kättesaadavale infole on arvutite ja andmebaaside ajastu tutvustanud uut tüüpi teadmisi, mida saab järeldada massiivsetes andmekogudes avalduvatest trendidest. Nii muutub aastate jooksul väärtuslikuks infokogumikuks andmebaas, kuhu salvestatakse igapäevased supermarketi külastajate ostud. Niisuguseid baase analüüsides saab teha mitmeid huvitavaid järeldusi tarbimisharjumuste või kliendigruppide kohta. Andmekaeveks (ingl. k. *Knowledge Discovery in Databases, Data Mining*) nimetatakse mittetriviaalset protsessi, mille käigus eraldatakse suurtest andmekogudest senitundmatut, kuid potentsiaalselt huvitavat informatsiooni [FPSM92]. Andmekaeve sai alguse 1990. aastate esimeses pooles kui interdistsiplinaarne uurimisvaldkond, mis kombineerib andmebaase, statistikat, tehisintellekti tehnikaid, tekstialgoritme jpm. suundi.

Suurte andmekogude analüüsis on mitmeid teravpunkte. Esimeseks ning kõige olulisemaks küsimuseks on andmete maht. Kaevandatavad kirjekogud on nii suured, et nende mahutamine arvuti mällu on raskendatud. Tihti on kogud salvestatud välistele mälukandjatele, mistõttu on oluline, et algoritmid töötaksid kiiresti ning võimalikult väheste läbivaatustega. Andmekaevealgoritmid peavad olema efektiivse mälutarbimisega, kuna vähimgi järeleandmine mälus toob kaasa protsessi aeglustumise või peatumise. Andmekaevealgoritmid peavad toime tulema suurte koguste ebaolulise infoga ning töötama ka puudulike või vigaste andmete korral. Kõikide võimalike kombinatsioonide uurimine on sageli ajalisest mahust tulenevalt välistatud. Andmete mahust lähtuvalt oleksid ideaalsed algoritmid, mis töötavad lineaarse ajaga.

Paljudes andmekaevealgoritmides on baaselementideks hulgad. Nii võib hulgana vaadelda näiteks ühe kliendi poolt poest korraga ostetud kaupu või teatavat tootegruppi tarbivaid inimesi. Kui vaadata kõiki võimalikke supermarketis saadavaid tooteid, moodustab ühe kliendi ostukorv sellest vaid suhteliselt väikese alamhulga. Hulkade vahesid ja ühisosasid kombineerides saab andmekogus identifitseerida olulisemaid reegleid või hoopis piirjuhte. Nii tekivad tarbijate grupeerimisel ostetud toodetest märgatavalt suuremad hulgad. Parimate gruppide leidmiseks tuleb võrdlusoperatsioone hulkadega sooritada väga palju.

Võib öelda, et hulgatehete efektiivsus on paljudes süsteemides võtmeküsimuseks. Kaeveprotsessi edukuse ja kiiruse seisukohast on oluline, et varieeruva suurusega hulgad oleksid mälu ja salvestusruumi suhtes efektiivsed ning operatsioonid hulkadega näitaksid head töökiirust.

1.1 Ülesandepüstitus

Käesoleva semestritöö eesmärgiks on võrrelda ja katsetada efektiivseid hulkade realiseerimise viise ning kiireid algoritme SPEXS-tarkvara baasil [Vil02]. Katsehulkadena kasutame nii sünteetilisi hulki kui ka reaalseid bioloogiliselt korreleeruvaid hulki. Samuti sisaldab töö ülevaadet hulgatehetel põhinevast andmekaevesüsteemist [KH95].

Hulkade realisatsioonidest vaatleme bitivektoreid ja täisarvude massiive. Töö võtmeküsimuseks on selgitada välja hulkade omadused, mil on efektiivne üht või teist realisatsiooni kasutada.

Semestritöö praktilise tulemusena valmis C++ programmides kasutamiseks mõeldud hulgatehete teek Libsets, testprogrammid ning Perlis kirjutatud abiprogrammid testhulkade genereerimiseks ja tekstikujul hulkade Libsets sisekujule teisendamiseks. Samuti valmis hulgatehteid kasutav programm geenifunktsioonide kaevandamiseks GO [go04] geeniontoloogiate baasist.

2 Hulgad ja realiseerimine

Hulk on matemaatikas üks lihtsamaid ja üldisemaid mõisteid. Mitteformaalselt võib hulgale viidata kui mingite objektide või mõistete kogule, millel on erinevalt ülejäänud objektidest või mõistetest mingi ühine omadus. Hulkadest rääkides ei saa mööda universaalhulga ja tühihulga mõistetest. Universaalhulk U ehk domeen sisaldab kõiki elemente antud kontekstis ning tühihulk $\oslash$ ei sisalda ühtegi elementi. Ostukorvi näidet jätkates on tühihulgaks tühi ostukorv ja domeeniks kõikide sellest kauplusest saadavate kaupade hulk. Edasises näeme, et hulkade realiseerimises on domeeni mõistel tähtis roll.

Tabel 1 sisaldab mõningaid olulisi hulgamõisteid ja tehteid.

$x \in A$	Element x kuulub hulka A
$\|A\|$	hulga A võimsus ehk temas leiduvate elementide arv
$\oslash$	Tühi hulk, $\| \oslash \| = 0$
U	Universaalhulk, domeen
$den\ (A)$	$\|A\|/\|U\|$, hulga tihedus (hõredus)
$A \cup B$	$\{x \| x \in A\ or\ x \in B\}$, hulkade A ja B ühend
$A \cap B$	$\{x \| x \in A\ and\ x \in B\}$, hulkade A ja B ühisosa
$A - B$	$\{x \| x \in A\ and\ x \notin B\}$, hulkade A ja B vahe
$A \otimes B$	$((A\ or\ B)\ not\ (A\ and\ B))$, hulkade A ja B sümmeetriline vahe
A^{-}	$(U\ not\ A)$, hulga A täiend
$A = B$	$\forall x \in U(x \in A \iff x \in B)$

Tabel 1: Hulkade mõisted ja operatsioonid

Hulgad võivad olla nii lõplikud kui lõpmatud. Lõpmatute hulkade korral võib veel rääkida loenduvatest ning mitteloenduvatest hulkadest. Käesoleva ülesandepüstituse raames tegeleme vaid lõplike hulkadega. Hulkade puhul on oluliseks

omaduseks elementide unikaalsus, st. hulk ei tohi sisaldada korduvaid elemente. Üldiselt ei ole hulga elemendid järjestatud. Hulgale sarnaseks mõisteks on hulkade hulk ehk multihulk (ingl. k. *bag*). Multihulgas on lubatud hulkade kordused.

Programmeerimise ja andmekaeve seisukohast on väga loomulikud positiivsete täisarvude hulgad. Arv võib tähistada näiteks andmebaasikirje võtit, viita mingile mälupiirkonnale, positsiooni jadas või stringis. Säilitades hulkades vaid viitasid või tegeliku info võtmeid, hoiame kokku nii hulkade kui ka vastavate operatsioonide kiiruse ning mäluvajaduse pealt. Samuti saame tagada hulgaelementide unikaalsuse nõude täitmise, kuna kirje võti või positsioon jadas on juba definitsiooni poolest unikaalne.

Täisarvude kasutamise hulgaelemendina teeb lihtsaks asjaolu, et programmeerimiskeeltes on täisarv erinevalt kirjest või stringist primitiivne andmetüüp, seega on elementide võrdsuse-võrratuse testimine lihtne ja kiiresti teostatav.

Järgnevas tutvume kahe võimaliku hulgarealisatsiooniga keeles C++.

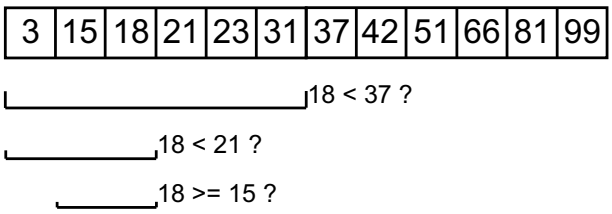
2.1 Täisarvude massiivid

Esimese hulgastruktuurina vaatleme täisarvude massiivi. Massiiv on hulga realiseerimiseks intuitiivseim lahendus. Keeles C saavad massiivielemendid alguse indeksist `0`. Hulga `S` i-nda elemendi poole pöördumiseks kasutame süntaksit `S[i]`. Viit massiivile `*S` on samaväärne selle esimese elemendiga `S[0]`. Täisarvude massiivi saame luua järgneva konstruktsiooni abil.

```
int S[5] = { 1, 8, 15, 72, 99 };
```

Kuigi hulkades pole definitsiooni kohaselt järjekord oluline, tasub tähele panna, et paljude algoritmide korral on siiski kasulik hulga elemente järjekorras hoida. Näiteks kahendotsingu algoritm (joonis 1) toimib nn. 'jaga-ja-valitse'-põhimõttel, milles jagatakse otsinguülesanne kasvavat (või kahanevat) järjekorda kasutades

väiksemateks osadeks ning elementi otsitakse rekursiivselt aina väiksemast piirkonnast. Jaga-ja-valitse printsiipi järgides on võimalik koostada mahukate ülesannete lahendamiseks paralleelseid algoritme.



Joonis 1: Kahendotsing täisarvumassiivides

Massiivide eeliseks teiste võimalike lahenduste ees on arusaadavus. Kasvavas järjekorras esitatud massiivid on loomulikud ja ka inimesele hõlpsasti loetavad.

Massiivina realiseeritud hulkadel on ka mitmeid puudusi. Esiteks ei saa keeles C++ ükski hulgaindeks ületada antud arhitektuuri suurimat võimalikku täisarvuväärtust `INT_MAX`. Tavaliselt on tänapäeval ülempiiriks $2^{31} - 1 = 2147483647$. Tegelik piir saabub aga palju varem. Kuna tavalise `int`-tüüpi täisarvu maht on mälus 4 baiti ning säilitatakse iga elemendi väärtust, võtab niisugune massiiv mälu 8Gb, mis käib enamikule arvuteist üle jõu.

Teiseks tuleb tähele panna, et massiiv on staatiline struktuur. Initsialiseerimisel luuakse etteantud suurusega massiiv ning eraldatakse sellele fikseeritud osa mälu. Massiivi suuruse muutmine ilma sisu ümber kirjutamata ei ole võimalik. Seega on mitmed hulgatehted, nagu ühendi leidmine või elemendi lisamine, üldjuhul samaväärsed uue massiivi loomise ja kõikide elementide uude massiivi kopeerimisega.

2.2 Bitivektorid

Hulgaloogika fundamentaalseim predikaat on test elemendi hulka kuuluvuse kohta, s.t. mingi domeeni iga elemendi ja vaadeldava hulga kohta on üheselt määratud, kas element on selle hulga liige või mitte. Osutub, et hulka saab kujutada

bitivektorina ning k-nda elemendi hulka kuuluvuse määrab bitivektori bitt k.

$$B = \{b_1, b_2 \dots b_n\},\ n = |U|,\ b_k = \texttt{true},\ kui\ k \in A,\ b_k = \texttt{false}\ vastasel\ juhul.$$

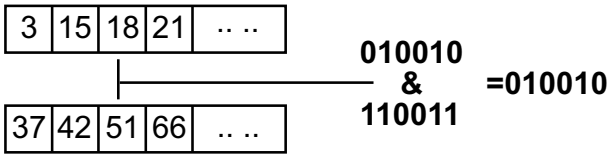
Keeles C++ saab bitivektoreid realiseerida näiteks täisarvumassiivina, kus iga arv esitub 4 baidi ehk 32 bitina. Nii on universaalhulka bitivektoritena 32 korda efektiivsem esitada kui lihtsalt täisarvude massiivina. Tuleb tähele panna, et bitivektorite korral säilitame infot `false` ka hulka mittekuuluvate elementide kohta, nii määrab bitivektori suuruse mitte hulka kuuluvate elementide arv, vaid universaalhulga U ehk domeeni suurus $|U|$. Seega peame alati mälu reserveerima terve domeeni jaoks ning hulga tiheduse vähenemisel väheneb bitivektori efektiivsus mälu suhtes.

Bitivektori realisatsiooni kavandamisel on ei tohi kahe silma vahele jätta asjaolu, et keeles C on tavaline täisarv märgiga, nn `signed int`. Seega on arvu maksimaalseks väärtuseks $2^{31} - 1$ ning märgi määrab suurima tähtsusega 32. bitt. Negatiivne täisarv `-i` esitatakse positiivse täisarvu `i+1` loogilise eitusena.

$$-4_{10} = 100_2 = not\ 3_{10} = not\ 11_2 = FFFFFFFC_{16}$$

Bitivektoritena hulkade kujutamisel on massiivide ees veel üks tugev eelis. Nimelt saame realisatsioonides kasutada C bititehteid (vt. tabel 2), mis on implementeeritud masinkoodilähedaselt ning peaks andma tunduvalt võitu töökiiruses.

Masina tasemel toimub bittide võrdlemine 4 baidi ehk ühe `int` kaupa, nagu on näha järgnevast ühisosa võtmise algoritmist joonisel 2.



Joonis 2: Ühisosa operatsioon bitivektorites

`&, &=`	and, bitikaupa korrutamine ja omistamine
`|, |=`	or, bitikaupa liitmine ja omistamine
`∧, ∧=`	xor, bitikaupa välistav liitmine ja omistamine
`», »=`	Bitikaupa paremale nihutamine ja omistamine
`«, «=`	Bitikaupa vasakule nihutamine ja omistamine

Tabel 2: C bitioperaatorid

2.3 Teek Libsets

Tarkvaraarenduse juures on paratamatu, et aja jooksul muutub programm aina mahukamaks ning selle linkimine ja kokkukompileerimine vaevanõudvamaks. Teatud etapis tasub mõtlema hakata tarkvara modulaarsuse ning taaskasutatavuse peale. Lahendusena on keeles C olemas teekide (ingl. k. *library*) mõiste. Teegiks nimetatakse binaarfaili, mis sisaldab endas mitmeid objektifaile ja mida kasutatakse programmi linkimisel ühtsena. Teegifail on reeglina indekseeritud, mistõttu on sellest erinevate sümbolite (funktsioonide, muutujate jms) leidmine kiirem kui objektifaile eraldi kompileerides.

Teeki Libsets koondasin kokku SPEXS-tarkvaras [Vil02] kasutatavad hulgaklassid. Libsets toetab hulgarealisatsioonidena bitivektoreid (klass `BA_Sets`) ja täisarvude massiive (klass `L_Sets`). Lisaks on loodud üldistav klass `Sets`, milles sooritatakse tehteid vastavalt implementatsioonile. Mõningaid operatsioone saab kasutada ka juhul, kui operandid on erinevatest tüüpidest. Tehnilist infot hulgateegi kompileerimise ja funktsionaalsuse kohta leiab töö lisadest.

Bitivektorid ja täisarvude massiivid on realiseeritud C++ massiividena. Esimesele kohale ehk indeksile `S[0]` on paigutatud hulga S elementide arv $|S|$. Bitivektorite eristamiseks on tähistatakse suurust negatiivsena. Bitivektorite loomisel tuleb arvestada domeenisuurusega `Domain_size`, massiivide korral on konst-

ruktori parameetriks hulga elementide arv `Size`. Binaarsed hulgatehted, mille tulemuseks on uus hulk (näiteks ühisosa), on realiseeritud kolmeoperandiliste funktsioonidena. Kolmanda operandi jaoks tuleb enne tehet mälu eraldada ning sellesse kirjutatakse tehte tulemus.

Operatsioonide juures tekitab probleeme kasutatava struktuuri staatilisus. Bitivektorite juures väljendub see asjaolus, et jooksvalt ei ole võimalik domeeni suurust muuta. Täisarvude massiivide korral on realiseerimata ühendi operatsioon, elemendi massiivi lisamine ning elemendi eemaldamine. Need operatsioonid on ajaliselt samaväärsed uue massiivi loomise ja elementide kopeerimisega.

Antud teegi versioonis on bitivektorites toetatud Little-Endian baidijärjestus, mille korral on vähimal mäluaadressil vähima tähtsusega element s.t. 0. Selline järjestus on standardne x86 arhitektuuri jaoks [osd04].

Teek Libsets kasutab sisendis ja väljundis hulkade tähistamiseks spetsiaalset süntaksit. Esimesel kohal on hulga elementide arv, mis bitivektorite korral on antud negatiivsena. Järgneb nurksulgudes komadega eristatud elementide või bitivektorite jada, kusjuures nurksulgudes on esimesel kohal `L:` või `B:` tähistamaks hulga tüüpi.

```
|S|<L:,e1,e2 ...en>

-|S|<B:,b1,b2 ...bn>
```

Lisaks hulgatehetele ning eelpool kirjeldatud kujul sisend-väljundile toetab teek erinevat tüüpi hulkade vahelist teisendamist, samuti on kaasatud multihulkade klass `Set_of_Sets` ning mitmesugused abiklassid, näiteks `error.c` veateadete töötluseks ning `infrastructure.c` ajakulu mõõtmiseks.

3 Hulkade efektiivsus

Selles peatükis kasutan teeki Libsets võrdlemaks bitivektorite ja massiivide efektiivsust erinevate sisendandmete korral. Võrdluseks olen loonud mõned testprogrammid ja abiskriptid, millest lähemat infot leiab töö lisadest.

Nagu eelnevas kirjeldasin, saab bitivektorit kasutades ühe `int`-tüüpi täisarvuga ehk 4 baidiga säilitada 32 järjestikust hulga elementi. Kirjeldades nii kõikvõimalikke domeeni U elemente täisarvudega $\{0, 1 \ldots |U| - 1\}$ võtavad bitivektorid kuni 32 korda vähem mälu kui massiiv. Siinkohal tuleb arvestada, et kui $|U| mod(32) > 0$, tuleb viimaste bittide jaoks ikkagi varuda täisarvu jagu mälu.

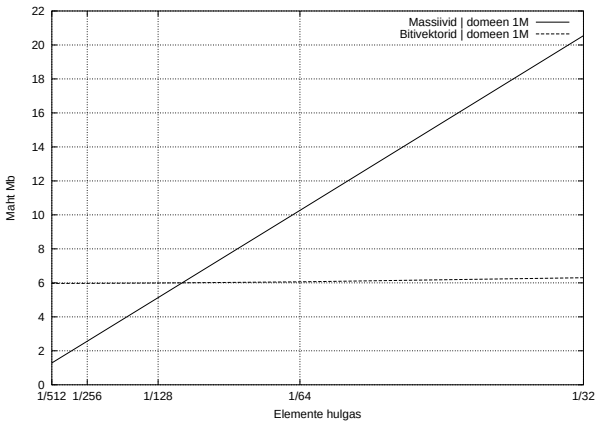
Samas, mida väiksem on bittidena kujutatud hulk võrreldes kõikvõimalike elementide hulgaga, seda suurem osa reserveeritud mälust kulub hulgast puuduvate elementide tähistamiseks bitiga `false`. Seega võib teoreetilise efektiivsuspiirina vaadelda arvu $\frac{1}{32}$. Kuna piir on mälus rangelt paigas, ei ole käesolevas töös uuritud mälukasutuse näitajaid. Operatsioonide ajakulu oleneb paljuski arhitektuurist ning on seetõttu peamiseks huvipunktiks.

Järgnevates alajaotustes uurin ja võrdlen bitivektorite ja massiivide käitumist erinevate sünteetiliste ja reaalsete bioloogiliste hulkade korral. Bioloogiliste hulkade taustast tuleb lähemalt juttu geenifunktsioonide kaevandamise peatükis. Sünteetilisi hulki kasutan salvestusruumi võrdlemisel, kuna neid saan genereerida meelevaldse suuruse ja tihedusega. Bioloogilised hulgad on kasutuses operatsioonide testimisel. Operatsioonide testimiseks on reaalsed hulgad paremad, kuna need on sageli üksteisega korrelatsioonis ning annavad ühisosade ja vahede leidmisel mitmekesisemaid tulemusi. Juhuslikud hõredad hulgad annavad aga ühisosa tulemusena suure tõenäosusega tühihulga $\oslash$.

Testid on teostatud AMD Athlon 1GHz protsessoriga masinal, mille operatiivmälu maht on 384Mb. Operatsioonisüsteemiks on Fedora Yarrow Linux 2.4 kerneliga.

3.1 Salvestusruumi võrdlus

Esimesena võrdlen teegi Libsets formaadis hulkade salvestamist tekstifailis. Vastav formaat on kirjeldatud jaotuses 2.3. Selleks otstarbeks genereerisin bitivektorite ja massiividega 5 faili. Igas failis on 100 hulka. Hulkade elemendid võivad olla vahemikus $\{0 \ldots 1'000'000\}$. Failides on hulgad eksponentsiaalselt kahanevate tihedustega $\frac{1}{32}$, $\frac{1}{64}$, $\frac{1}{128}$, $\frac{1}{256}$, $\frac{1}{512}$.

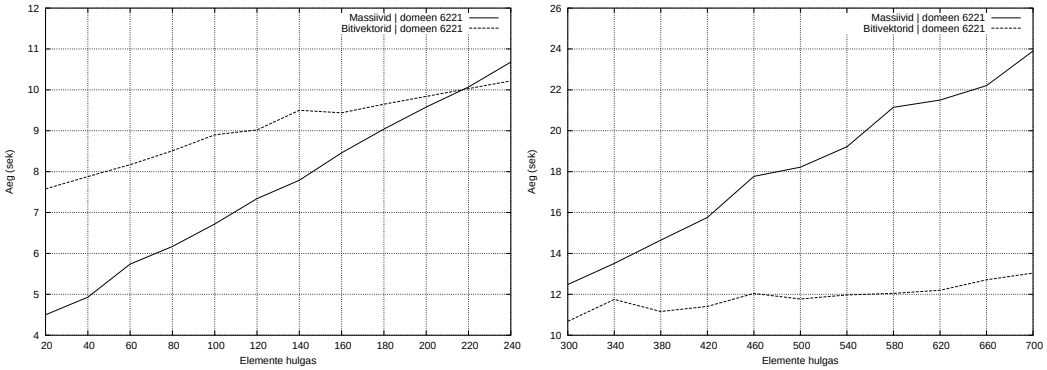


Joonis 3: Hulkade salvestusruum tekstifailis erinevate tiheduste korral

Joonisel 3 Näeme, et bitivektorite ruumikulu on stabiilselt 6Mb juures. Bitivektorite juures on näha vaid õige väikest tõusu tiheduse kasvades, on tingitud kümnendkujul olevate vektorite tihenemisest. Massiivide ruumikulu tõuseb hulkade tihenedes lineaarselt pooleteistkümnelt megabaidilt kahekümneni. Kuigi iga täisarv esindab bitivektorites 32 hulgapositsiooni, tuleb tähele panna, et failides kasutatakse hulgaelementide eraldajateks komasid ning bitivektori esituses märgime lisaks kõikidele domeeni tühjadele elementidele ka eraldajaid, mis kujutab endast positsiooni kohta ühte baiti lisakulu. Massiivid muutuvad bitivektoritega võrreldes ruumitarbe osas tiheduse kasvades efektiivsemaks ligikaudu tihedusest $\frac{1}{120}$.

3.2 Sisendi kiiruse võrdlus

Järgmisena esitan võrdluse bioloogiliselt korreleeritud hulkade sisendist lugemise kiiruste kohta. Selleks kasutan kahte 12-st failist koosnevat testfailide kogu väiksemate ja suuremate hulkade tarbeks. Esimeses kogus on hulgad elementide arvuga $\{20, 40 \ldots 240\}$, teises $\{300, 340 \ldots 700\}$. Kõikide hulkade domeen on 6221, samuti on igas failis 6221 hulka.



Joonis 4: Sisendi kiirus erinevate tiheduste korral

Ka selle graafiku juures (joonis 4) on näha bitivektorite suhteliselt stabiilset esinemist võrreldes massiividega. Kuigi võiks arvata, et bitivektorite ajanõue hulkade tihenedes ei muutu, on graafikutelt siiski märgata mõningast kasvu. Kasvu võib põhjendada faktiga, et kümnendesituses on tihedad bitivektorid keskmiselt pikemad ning nende sisendist lugemine bitikaupa nihutamise operaatori » abil aeganõudvam. Ajafunktsioonide lõikepunkt on ligikaudu 215 juures. Seega võib öelda, et sisendi suhtes on bitivektorid efektiivsemad, kui hulgas on vähemalt $\frac{1}{32}$ domeeni elementidest.

```
for ( int i = 0; i < sizeof(test_set); i++ ){
  i_set s1 = test_set[i];
  for (int j = i+1; j < no_of_sets; j++){
    i_set s2 = test_set[j];
    i_set tulemus = create();
    subtract ( s1, s2, result );
    intersect ( s1, s2, result );
    del_set ( tulemus );
  }
}
```

Joonis 5: Operatsioonide testskeem

3.3 Operatsioonide kiiruse võrdlus

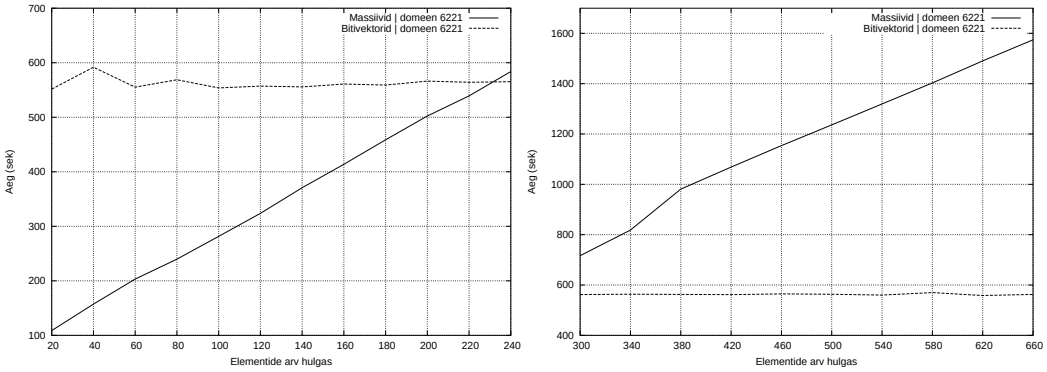
Operatsioonide testimiseks kasutan kahte lähenemist. Esmalt rakendan eelmises ülesandes kasutatud testkomplekte ning võrdlen operatsioonide sooritamise kiirust bitivektorite ja massiivide vahel.

Teiseks testiks olen loonud 6221-hulgalise faili, mis sisaldab erinevaid täisarvuhulki pikkustega {10, 20, . . . 100, 300, 340 . . . 700}. Teise testi sisuks on hulgaoperatsioonide kiiruste võrdlemine, kui suuremad täisarvumassiivid teisendatakse enne tehete sooritamist bitivektoriteks. Teisenduspiiri tähistav n.n. teisenduskoefitsent varieerub vahemikus [0. . . 10%] 0.25% intervalliga ning tähistab piiri, millest suuremad hulgad (võrreldes domeeniga) teisendatakse enne tehteid bitivektoreiks. Kui koefitsent on null, teisendatakse kõik hulgad bitivektoreiks. Ajatulemustesse ei ole sisse arvatud sisendile ja teisendustele kulunud sekundeid.

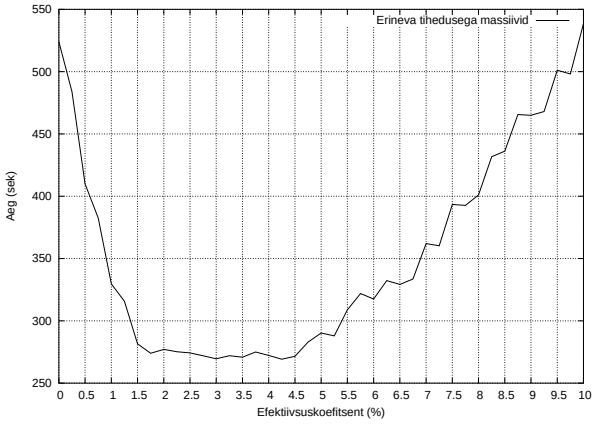
Mõlemas testis järgitakse üldjoontes samasugust skeemi (v.t. joonis 5). Esimeses testis sooritatakse tehteid sama tüüpi hulkade vahel (esindajad klassidest `BA_Sets` ja `L_Sets`). Teise testi juures kasutatakse üldistavat klassi `Sets`. Et-

teantud n hulga korral sooritatakse $n * (n - 1)$ tehet ja leitakse kõikide hulkade omavahelised ühisosad ja vahed. Iga tsükli alguses loon tühja resultaathulga ning peale ühisosa ja vahe leidmist vabastan uuesti mälu. Ilma mälu vabastamata muutusid ajavahed saalimise tõttu suuremate hulkade korral talumatult pikaks.

Testida oleks võinud ka mõlemaid tehteid eraldi, kuid usun, et piisava ülevaate efektiivsusest saab ka mõlemat operatsiooni järjestikku rakendades.



Joonis 6: Operatsioonide kiirus erinevate tiheduste korral



Joonis 7: Operatsioonide kiirus erinevate teisenduskoefitsentide korral

Esimeses testis suurte ja väikeste hulkade operatsioone uurides sain eelnevatele võrdlustele üsna sarnased tulemused (v.t. graafik joonisel 6). Bitivektorite

töökiirus püsib stabiilsel tasemel, väikeste hulkade esimeses pooles sissesattunud järsem kurv on ilmselt juhuslik. Täisarvumassiivide töökiirus kasvab lineaarselt ning ületab bitivektorite efektiivsuspiiri ligikaudu teoreetiliselt põhjendatud tasemel $\frac{1}{32}$.

Teise testi tulemuste graafikult (Joonis 7) näeme, et parimat eelistuspunkti on raske määrata, pigem on tegu efektiivsusvahemikuga. Sellise tulemuse juhuslikkus on küllalt ebatõenäoline, kuna kordasin testi ka teises masinas ning sain analoogilise kujuga kõvera. Pigem võib olla tegu konkreetse bioloogiliselt korreleeritud andmestiku eripäraga. Graafiku põhjal võime järeldada, et hulgatehted on efektiivseimad, kui bitivektoritena kasutatakse hulki, milles on mitte vähem kui 1.5% ja mitte rohkem kui 4.5% kõigist elementidest. Murdarvudena kujutatud ligikaudse vahemiku $(\frac{1}{66}, \frac{1}{22})$ aritmeetriline kese $\frac{1}{44}$ on teoreetilisest piirist $\frac{1}{32}$ veidi nihkes, kuid annab sellele piisavalt täpse kinnituse.

4 Sagedaste alamhulkade kaevandamine

Järgnevalt vaatlen hulgaoperatsioonidel põhinevat rakendust andmekaeves. Rakenduse eesmärgiks on andmebaasi kirjetest omaduste järgi sagedaste alamhulkade leidmine ning alamhulkade statistiliste omaduste abil andmekogus kehtivate reeglite ennustamine.

Sagedaste alamhulkade ja vastavate reeglite genereerimise vajalikkusest võib tuua järgneva näite. Vaatleme liikluskindlustuskompanii klientide andmebaasi. Kindlasti on huvipakkuv reegel, mis määrab klientide kohta liiklusõnnetuse toimumise tõenäosuse. Iga üksiku kliendi jaoks ei ole toimumistõenäosuse arvutamine võimalik, kuid tõenäosust saab leida teatavate klientide omaduste, st alamhulkade kaudu. Näiteks on teada, et noortel meestel on kombeks sattuda tihemini avariidesse kui teistel sõitjagruppidel.

Kersteni ja Holsheimeri [KH95] poolt kirjeldatud süsteem koosneb kahest komponendist - andmekaevurist ja andmelaost. Andmekaevuri ülesandeks on päringute formuleerimine, kaeveprotsessi juhtimine ja reeglite tuvastamine, andmelaokomponent juhib päringute täitmist ning tagastab vastava kirjete hulga või hulga kohta kehtivat statistilist infot.

4.1 Andmekaevur

Esimesena vaatleme andmekaevuri tööd. Andmekaevuri ülesandeks on kaeveprotsessi juhtimine reeglite loomise teel. Andmebaas koosneb klientide kirjetest {K}, iga kirje aga omakorda atribuutidest $\{A_1, A_2, A_3 \ldots A_n\}$, näiteks kliendi vanus, sugu, kodulinn, eelnev osalemine liiklusõnnetuses, jms. Atribuutidel võib olla mitmesuguseid väärtusi $\{V_j\}$, väärtused võivad olla binaarsed, numbrilised kui ka nominaalsed. Kasutaja poolt määratakse üks atribuut A_Z, nn sihttunnus (ingl. k. *target attribute*). Käesoleva töö raames on sihttunnuseks binaarne tunnus. Andme-

kaevuri ülesandeks on tunnuste kombineerimine tõenäosuslikeks reegliteks järgneval kujul.

$$\{A_{i_1} = V_{j_1}, A_{i_2} = V_{j_2} \ldots A_{i_k} = V_{j_k}\} \Rightarrow A_Z = \texttt{true}\ (v\tilde{o}i\ A_Z = \texttt{false}) \mid P.$$

Andmebaasi ülesandeks on siis vastavate päringute täitmine või agregaatinfo leidmine. Kindlustusfirma võiks olla huvitatud näiteks järgneva reegli leidmisest, mis on leitud baasist nimetatud tunnuste kombineerimisel.

$$\{Sugu = M,\ vanus \in [19, 24]\} \Rightarrow \tilde{o}nnetus = \texttt{true} \mid P = 61.3\%$$

Reeglite loomisel kasutatakse iteratiivset tehnikat, mille abil defineeritakse algreegel (ingl. k. *initial rule*) ning hakatakse sellele lisapäringute abil laiendusi (ingl. k. *extensions*) genereerima. Laiendusi genereeritakse seni, kuni ei saa enam ühegi tunnuse abil laieneda või on ületatud etteantud otsingusügavus (iteratsioonide arv). Algsesse reeglisse valitakse tühi eeldus, mis näitab, kui suurel osal kõikidest kirjetest on vaadeldav sihttunnus. Olgu näiteks pooltel kindlustusklientidel toimunud õnnetus.

$$\{\texttt{true}\} \Rightarrow \tilde{o}nnetus = \texttt{true} \mid P = 50.0\%$$

Nüüd vaatleme kõiki õnnetusega noori kliente, s.t loome reeglile laienduse. Tuleb välja, et kõikidest 19-24 aastastest on õnnetusi rohkem kui pooltel. Uue reegli katteks nimetatakse uue alamhulga suurust, näiteks siis kõikide noorte klientide arvu.

$$\{Vanus \in [19, 24]\} \Rightarrow \tilde{o}nnetus = \texttt{true} \mid P = 55.4\%$$

Üldiselt ei huvita meid mitte kõikvõimalike gruppide õnnetustõenäosused, vaid ainult piirjuhud - kõige ettevaatlikumad ning kõige ohtlikumad juhid. Seetõttu on sobilik mäkkeronimise (ingl. k. *hillclimbing*) tehnika, mille korral genereeritakse olemasolevale reeglile kõikvõimalikud laiendused, kuid säilitatakse

laiendus, mille korral on muutus meid huvitavas suunas (tõenäosuses) kõige suurem. Antud algoritmi on võimalik ka edukalt paralleelselt täita. Nn. kiirteotsingu abil (ingl. k. *beam-search*) kasutatakse korraga n mäkkeronijat. Iga iteratsiooni järel valitakse tekkinud reeglite hulgalt jätkamiseks n parimat reeglit.

Järgnev näide (tabel 3) illustreerib reeglite iteratiivset laiendamist etappide kaupa. Tasub tähele panna, et iga laiendusega kitseneb vaadeldavate kirjete hulk. Teiseks on oluline asjaolu, et uutel etappidel saab taaskasutada eelmiste etappide tulemusi. Olulised tulemused on saadud tabelis teisel ja kolmandal real olevate reeglite laiendamisel.

Etapp 0	Etapp 1	Etapp 2	Etapp 3
`true` $50.0\%(0.0\%)$	Vanus[19,24] $55.4\%(5.4\%)$		
	Sugu N $46.3\%(-3.7\%)$	**Vanus[30,33]** $44.2\%(-5.8\%)$	
	Sugu M $53.2\%(3.2\%)$	**Vanus[19,24]** $61.3\%(11.3\%)$	
	Tüüp liising $51.3\%(1.3\%)$	Vanus[19,24] $57.7\%(7.7\%)$	Sugu N $52.0\%(2.0\%)$
	Vanus[25,33] $48.7\%(-1.3\%)$	Sugu N $45.3\%(-4.7\%)$	

Tabel 3: Reeglite iteratiivne laiendamine

4.2 Andmeladu

Hulgastruktuuride ja operatsioonide peamine rakendus avaldub vaadeldava süsteemi andmelaokomponendis. Klientide andmestu on kujutatud binaarsete assot-

siatsioonitabelitena ehk BAT-idena (ingl. k. *binary association tables*). Andmelaokomponenti võib vaadelda kui relatsioonilist andmebaasi, mille igas tabelis on kujutatud klientide identifikaatorid ja ühe atribuudi väärtused. Näiteks esitan binaarse assotsiatsioonitabeli atribuudi 'sugu' kohta.

$$\{(\texttt{15 M}),\ (\texttt{17 N}),\ (\texttt{75 M}),\ (\texttt{238 N}),\ (\texttt{666 N})\}$$

Mõned atribuudid, näiteks 'vanus', on esitatud klientide hulga suhtes tihedatena, s.t BAT-is on esindatud iga kliendi täpne vanus. Samas on 'auto hind' esitatud vahemikena, seega on erinevaid väärtusi klientide arvuga võrreldes vähe ja hulk hõre. Mittebinaarsete tunnuste teisendamine binaarseteks toimub optimaalsete vahemike defineerimise abil. Vastavad algoritmid jäävad antud töö kontekstist välja. Binaarne sihttunnus A_Z 'õnnetus' on esindatud kahe tabeliga, millest ühes on liiklusõnnetuses osalenud kliendid ning teises õnnetuseta kliendid. Niisuguste BAT-idega opereerimiseks on näidatud mõned lisaoperatsioonid tabelis 4.

AB	$\{ab \mid a\ on\ kirje\ võti\ \cap\ b\ on\ tunnuse\ väärtus\}$
$AB.select(v_a, v_ü)$	$\{ab \mid ab \in AB\ \cap\ b \geq v_a\ \cap\ b \leq v_ü\}$
$AB.select(v)$	$\{ab \mid ab \in AB\ \cap\ b = v\}$
$AB.semijoin(CD)$	$\{ab \mid ab \in AB\ \cap\ \exists cd \in CD\ \cap\ a = c\}$
$AB.histogram()$	$\{bf \mid ab \in AB\ \cap\ f\ on\ b\ esinemissagedus\}$

Tabel 4: Hulgatehted binaarsete assotsiatsioonitabelitega (BAT)

Nagu eelnevas kirjeldasime, toimub reeglite leidmine etappide kaupa. Igas etapis proovitakse olemasolevale reeglile lisada vaadeldava atribuudi erinevaid väärtusi või vahemikke, näiteks soo korral väärtusi $\{M, N\}$, vanuse korral $\{[19, 20] \ldots\}$. Paneme tähele, et reegli täiendamisel lisame ainult neid atribuute, mis juba olemasolevas reeglis ei sisaldu. Niimoodi avaldub nn vertikaalsuumi (ingl. k. *vertical*

zoom) omadus: otsingu edenedes huvipakkuvate atribuutide (ja reeglite võimalike laienduste) arv väheneb.

Peamiseks probleemiks ja kuluallikaks kujuneb kõikvõimalike laienduste kvaliteedi ehk tõenäosuse arvutamine. Intuitiivseks lahenduseks oleks iga võimaliku laienduse korral teha 2 päringut: esimeses leitakse kirjed $\{K^+\}$, mille korral sihtatribuut on tõene, teises kirjed $\{K^-\}$, mille korral sihtatribuut on väär, ning uue reegli hindamiseks leitakse suhe $\frac{|\{K^+\}|}{|\{K^+,K^-\}|}$. Selgub aga, et niisugune lähenemine on liiga töömahukas ning saab ka paremini.

Kujutame süsteemi nii, et andmekaevur esitab andmelaole päringuid sageduste kohta, kombineerides päringusse ühe vaadeldava atribuudi väärtusi ning sihtatribuudi väärtusi. Paljude (n) võimalike väärtuste korral, näiteks vanus või kodulinn, teisendatakse atribuudid binaarseteks $2n$ klassi arvutamise teel ning leitakse vastavad sagedused.

Näeme, et BAT-e kasutades taandub sagedaste alamhulkade otsing sisuliselt hulgaaritmeetikale. Mäletatavasti hoiti BAT-ides objektide idente ja ühe atribuudi väärtusi, sihtatribuudi korral loodi aga kaks tabelit tõeste ja väärade väärtuste hoidmiseks.

Esimesel sammul on katteks terve baas ning võimalikud paarid (näiteks (sugu, õnnetus)) on $\{(M,+),(M,-),(N,+),(N,-)\}$. Leiame järgnevad ajutised sagedustabelid erinevatele atribuutidele.

$$T_0^+ = semijoin(terveTabel\_sugu, \tilde{o}nnetus^+).histogram$$
$$T_0^- = semijoin(terveTabel\_sugu, \tilde{o}nnetus^-).histogram$$
$$T_1^+ = semijoin(terveTabel\_autohind, \tilde{o}nnetus^+).histogram$$
$$T_1^- = semijoin(terveTabel\_autohind, \tilde{o}nnetus^-).histogram$$
$$\dots$$

Tulemusteks saadakse õnnetustõenäosused erinevate tunnuste lõikes, mis tagastatakse andmekaevurile. Histogrammide põhjal tehakse otsus, millises suunas

mäkketõusu jätkata.

$T_0^+ = \{(M, 53.2\%), \ \ (N, 46.3\%)\}$

$T_0^- = \{(M, 46.8\%), \ \ (N, 53.7\%)\}$

$T_1^+ = \{(hind < 100K, 50.4\%), \ \ (hind > 100K, 49.8\%)\}$

...

Järgmisel sammul tuleb hinnata mingi arvu laienduste headust. Leitakse alamhulgad kombineerides sihtatribuuti 'õnnetus' A_Z teiste võimalike atribuutidega.

$Tmp_0 = terveTabel\_vanus.select(19, 24)$

$\tilde{o}nnetus_0^+ = semijoin(Tmp_0, \tilde{o}nnetus^+)$

$\tilde{o}nnetus_0^- = semijoin(Tmp_0, \tilde{o}nnetus^-)$

$Tmp_1 = terveTabel\_sugu.select('N')$

$onnetus_1^+ = semijoin(Tmp_1, \tilde{o}nnetus^+)$

$onnetus_1^- = semijoin(Tmp_1, \tilde{o}nnetus^-)$

$Tmp_2 = terveTabel\_sugu.select('M')$

...

Seejärel uuritakse, kuidas käituvad õnnetuste sagedused, kui laiendada reeglit. Paneme tähele, et reeglit laiendatakse vaid nende atribuutide arvel, mis juba olemasolevas reeglis ei sisaldu. Samuti avaldub siin horisontaalne suum (ingl. k. *horizontal zoom*). Näiteks tabel $\tilde{o}nnetus_0^+$ sisaldab avariis osalenud 19-24-aastasi juhte. Kui hakkame reeglile genereerima laiendust (sugu = 'M'), ei pea me enam läbi vaatama kõiki avariis osalenud juhte, piisab vaid noorte juhtide tabelist.

$T_0^+ = semijoin(terveTabel\_sugu, \tilde{o}nnetus_0^+).histogram$

$T_0^- = semijoin(terveTabel\_sugu, \tilde{o}nnetus_0^-).histogram$

$T_1^+ = semijoin(terveTabel\_sugu, \tilde{o}nnetus_3^+).histogram$

$T_1^- = semijoin(terveTabel\_sugu, \tilde{o}nnetus_3^-).histogram$

$T_2^+ = semijoin(terveTabel\_sugu, \tilde{o}nnetus_4^+).histogram$

$T_2^- = semijoin(terveTabel\_sugu, \tilde{o}nnetus_4^-).histogram$

$T_3^+ = semijoin(terveTabel\_autohind, \tilde{o}nnetus_0^+).histogram$

$T_3^- = semijoin(terveTabel\_autohind, \tilde{o}nnetus_0^-).histogram$

$T_4^+ = semijoin(terveTabel\_autohind, \tilde{o}nnetus_1^+).histogram$

...

Tulemuseks saadakse taas histogrammid, mis tagastatakse andmekaevurile, seejärel valitakse hulk parimaid laiendusi. Laiendamine kordub, kuni laiendusi ei saa rohkem genereerida, kuna kõik atribuudid on kaasatud. Samuti võib anda ette otsingusügavuse. Tulemuseks saame kogu andmehulgal kehtivaid huvitavaid seaduspärasusi.

5 Geenifunktsioonide kaevandamine

Tervete genoomide järjest kiirenev kaardistamine on mõjutanud eksperimentaalbioloogia teooriat ja praktikat. Esmane võrdlus *saccharomyces cerevisiae* ehk toidupärmi genoomi ja geeniuuringutes populaarse ussikese *caenorhabditis elegans* genoomi vahel viis üllatusliku avastuseni. Nimelt leiti, et 12% *C.elegans'* 12'000 geenist kodeerivad valke, mille bioloogiline funktsionaalus sarnaneb vastavatele pärmi geenide (ca 27% tollal teadaolevatest 5700-st) poolt kodeeritud valkudele [ABB00]. On leitud, et enamik sellistest valkudest on seotud bioloogiliste tuumprotsessidega, näiteks DNA-replikatsioon, transkriptsioon ja metabolism. Hilisem kolmepoolne võrdlus eelmainitute ja äädikakärbse *drosophila melanogaster* vahel on näidanud, et samal pärmigeenide alamhulgal on äratuntavad sarnasused ka äädikakärbse genoomis.

Niisugused järeldused on viinud tunnustatud arvamusele, et on olemas ühtne ja piiratud valkude ja geenide universum, millest paljud on esindatud enamikes või kõigis elusrakkudes ning nende geenide funktsionaalsus on üldjoontes sarnane. Seisukoht toidab bioloogia teaduslikku ühendamisprotsessi, mille käigus kaardistatakse ühiseid geene ning otsitakse funktsionaalsuse analoogiaid mitmekesiste organismide vahel. Geenisekventside ja funktsioonide oluline sarnasus pakub teaduslikust perspektiivist mitmeid väljakutseid. Näiteks tuleb uurida võimalusi kandmaks lihtsamate organismide eksperimentaaltulemustest saadud mõistete struktuure üle keerukamate organismide juurde.

Paraku ei suuda ühendamisprotsess sageli teadusandmete juurdevoolu kiirusega sammu pidada. Vaatamata levinud seisukohale geenide funktsionaalsuse ühtsusest valitseb molekulaarbioloogia andmebaasides terminoloogiline segadus.

5.1 Geeniontoloogiate andmekogu GO

Projekt Gene Ontology (GO) [ABB00, HCI04] on loodud edendamaks erinevate molekulaarbioloogia andmebaaside vahelist integratsiooni. GO eesmärgiks on geenisekventside, nende omaduste ja vastavate valkude klassifitseerimise standardimine ontoloogiateks.

Ontoloogiaks nimetatakse hästidefineeritud mõistete kogu või terminoloogiasõnastikku (ingl. k. *vocabulary*), milles kehtivad mõistetevahelised reeglid. GO terminid on kujutatud unikaalsete identifikaatoritena (näiteks `GO:0018282`), millega on seotud tekstiline kirjeldusväli. Ontoloogiad toetavad hierarhilisi seoseid `is-a` ja `part-of`. Ontoloogiaid võime kujutada suunatud atsüklilise graafina, milles üldisemad terminid on seotud spetsiifilisematega (graafis alluvate tippudena) ning igal alluval võib olla mitu eellast.

GO projektis kirjeldavad ontoloogiad kolme omavahel mittelõikuvat molekulaarbioloogia domeeni - molekulaarfunktsiooni, biolooglist protsessi ja rakukomponenti. Need kolme domeeni elemendid on geenide, kodeeritud valkude või valgugruppide atribuutideks.Bioloogilist reaalsust peegeldab asjaolu, et iga geeniga või valguga võib olla seotud mitu terminit. Samuti võib iga terminiga siduda 0 või enam geeni, seega on tegemist ’m-n’ seosega.

GO projekt sai alguse 1998. aastal äädikakärbse, hiire ja pärmi genoomiprojektide koostööna. Käesolevaks ajaks on projektiga ühinenud paljud olulised genoomibaasid. Ligi pooltele pärmi geenidele pole seni funktsionaalsust määratud. Andmekaeve perspektiivist pakub meile huvi võimalused kasutamaks GO ontoloogiaid geenide funktsioonide ennustamiseks.

5.2 Geenide klasterdamine

Erinevates eksperimentaaltingimustes sooritatud geenieksressiooni mõõtmiste abil on võimalik kirjeldada vaadeldavate geenide ekspressiooniprofiili. Hüpoteesi kohaselt võib sarnaste ekspressiooniprofiilidega geenidel (ehk koosavalduvatel geenidel) olla ühiseid jooni regulatsioonimehhanismides. Seega võime sarnaste ekspressioonitasemetega geene klasterdades tuvastada sarnaselt reguleeritavaid geenide gruppe, mis omavad potentsiaalselt sarnast funktsionaalsust.

Objektide grupeerimist sarnaste objektide klassideks nimetatakse klasterdamiseks (ingl. k. *clustering*). Klaster (ingl. k. *cluster*) on objektide kogum, mis on sarnased antud klastri sees ning erinevad teiste klastrite objektidest [Juh03]. Klasterdamise puhul on tähtis hea sarnasuse meetrika olemasolu, mille põhjal erinevaid objekte omavahel sarnasteks või erinevateks lugeda. Levinumateks klasterdamismeetoditeks on K-keskmise meetod ja K-metoidi meetod. K-keskmise meetod saab ette klastrite arvu k ning sarnasuse mõõduna kasutab klastrisse kuuluvate elementide aritmeetrilist keskmist. K-metoidi meetodi erinevus seisneb selles, et iga klastri keskmeks valitakse üks olemasolev element, mille ümber hakatakse sarnaseid elemente koondama.

Käesolevas eksperimendis rakendasin andmeid, mida on kirjeldatud allikas [VBJ$^+$00]. Testandmeteks on 6221 toidupärmi geeni, mille ekspressiooniprofiilid on kirjeldatud 80 bioloogilise katse raames. Seega on iga geen vaadeldav kui 80-elemendine vektor. Antud geenide hulgalt on leitud 6221 klastrit. Iga klastri keskpunktiks on valitud järjekordne geen ning järgnevad geenid on järjestatud vastavalt sellele, kui sarnane on nende ekspressiooniprofiil esimese geeni profiilile 80 eksperimendi lõikes. Kauguse mõõduna on kasutatud koosinuskaugust 80-mõõtmelises ruumis. Kahemõõtmelises ruumis on kahe vektori v_r ja v_s vaheline koosinuskaugus d_{rs} arvutatav järgnevalt.

$$d_{rs} = 1 - \frac{x_{r_x} x_{s_y}}{(x_{r_x} x_{r_y})^{\frac{1}{2}} (x_{s_x} x_{s_y})^{\frac{1}{2}}}$$

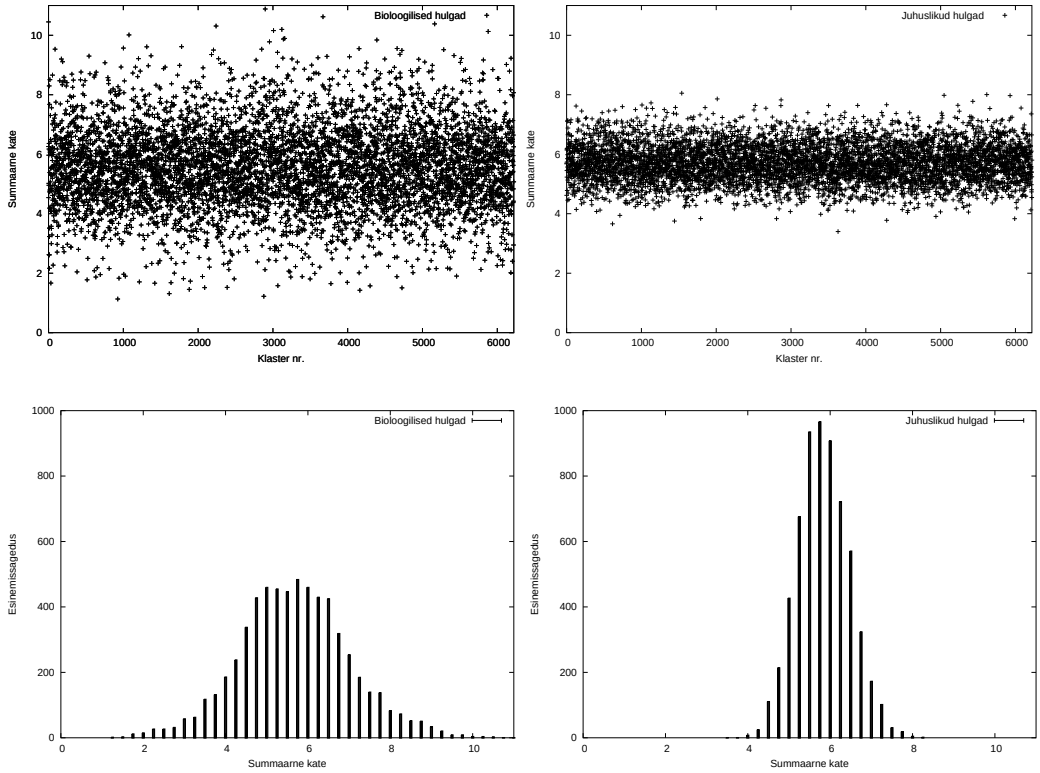
Klasterdamise tulemusena on eksperimendi testandmeteks $(6221 * 6221)$ maatriks ehk 6221 bioloogilises mõttes olulisimat järjestust või klastrit, kusjuures paljud nendest järjestusest on suures osas kattuvad. Meenutame, et 6221 elementi on võimalik järjestada kokku $6221!$ moel, seega vaatleme ainult $\frac{1}{6220!}$ võimalikest järjestustest. Nendest hulkadest sobiva pikkusega alamhulki moodustades võib oletada, et osadel klastritest tekivad GO ontoloogiatesse lahterdatud geenihulkadega olulised ühisosad. Statistiliselt optimaalse alamhulga pikkuse valimise metoodikad jäävad antud töö kontekstist välja, see jäetakse programmi kasutaja katsetada ja valida. Tegemist ei ole just klassikalise klasterdusmeetodiga, kuid analoogilised eksperimendid reaalsetel bioloogilistel andmetel on end juba varem õigustanud.

5.3 Eksperiment

Eksperimendi sisendandmeteks on ühelt poolt GO kategooriatesse annoteeritud 3068 pärmi geeni. Need on Perli skriptide abil teisendatud Libsets hulgateegi poolt loetavasse vormingusse. Iga hulk on seega ühte vaadeldavasse kategooriasse kuuluv geeniidentifikaatorite hulk.

Teisalt olen 6221. viisil klasterdatud geenidest võtnud meelevaldse arvu esimesi elemente. Käesolevas eksperimendis on selleks arvuks 100. Võrdluseks tekitasin 6221 juhuslikku 100-elemendilist hulka, mille domeeniks on samuti 6221.

Eksperimendi sisuks on iga klastri korral uurida, kuidas tema 100 esimest elementi sobituvad GO kategooriatesse, s.t millised on ühisosad. Siinkohal tasub tähele panna, et iga võrreldav järjestus annab kindlasti suuri ühisosasid mõningate kategooriatega. Teame, et GO kategooriad on hierarhilised, seega näiteks juurkategooria alamhulgaks on kõik GO annoteeritud pärmigeenid.



Joonis 8: Hulkade ülekatted ontoloogiatega

Seetõttu kasutan leitud ühisosade headuse hindamiseks ülekatte mõõtu. Hulkade A ja B ülekatteks $c_{A,B}$ nimetatakse alljärgnevat suurust.

$$c_{A,B} = \frac{|A \; and \; B|}{|A \; or \; B|} = \frac{|A \; and \; B|}{|A| + |B| - |A \; and \; B|}$$

Ülekate jääb vahemikku $[0, 1]$ ning annab kõrgemaid tulemusi juhul, kui ühisosa moodustab mõlemast hulgast olulise osa. Eksperimendis kasutan klastri G_i headuse hindamiseks summaarse ülekatte C_{G_i} mõõtu.

$$C_{G_i} = \sum_{F_j \in GO} c(G_i, F_j)$$

Visuaalsel vaatlusel koonduvad mõlemad ülekatete jaotused normaaljaotuseks väärtuse 5 ümber. Samuti on näha teatav konstantne ülekate, mis esineb kõigi

kategooriate juures ja tuleneb ilmselt GO-kategooriate hierarhilisest struktuurist.

Võrreldes bioloogilises mõttes järjestatud geenide ning juhuslikult genereeritud hulkade poolt tekitatud ülekatete jaotusi, näeme, et juhuslike hulkade katted koonduvad palju tihedamalt oma telje ümbrusse ning vastavad ekstreemumkohad on tunduvalt väikesemad.

Bioloogiliselt järjestatud geenide ülekatted seevastu näitavad hajuvust ning suuri ekstreemume, millest võib järeldada, et geenijärjestuste hulgas on klastrite headus palju varieeruvam. Suurte ülekatte väärtuse korral on klastri keskpunktiks valitud geen, mis asub paljude sarnaste geenide läheduses ning vaadeldavat geenide hulka on võimalik kirjeldada mingite ühisnäitajatega, järelikult on tegemist hea kirjeldava klastriga. Väikesed ülekatte väärtused seevastu näitavad, et esimeseks valitud geen asub järgnevatest suhteliselt kaugel ning ühiseid nimetajaid ei leidu või on need statistiliselt piisavalt ebaolulised, näiteks sarnasus juurkategooriaga vms.

Antud eksperimendi nõrgaks kohaks on asjaolu, et esialgne järjestuse pikkus sai valitud meelevaldselt ning mõne teistsuguse algpikkuse puhul võivad tulemused tulla teistsugused. Kindlasti tuleks uurida statistilisi võtteid optimaalse pikkuse arvutamiseks. Samuti võib korrata eksperimenti erinevate väärtustega ning uurida edasi klastreid, mis annavad mitmete eksperimentide lõikes ülejäänutest silmapaistvalt paremaid tulemusi.

6 Kokkuvõte

Hulk on matemaatikas üks fundamentaalsemaid mõisteid. Suur osa programmeerimise ülesandepüstitustest on seotud hulkadega ning kiire hulgaaritmeetika realiseerimine on paljude algoritmide võtmeküsimuseks. Andmekaeves on hulgaoperatsioonid kasutusel näiteks andmekogude vaheliste sarnasuste otsimises.

Käesolevas semestritöös uurisin täisarvude massiive ja bitivektoreid kui kiireid hulgastruktuure ning võrdlesin hulkade sisendist lugemise, salvestusruumi ning ühisosa ja vahe leidmise efektiivsust erinevate sisendandmete korral. Töö järeldusena võib väita, et täisarvuhulkasid tasub realiseerida bitivektoritena, kui hulkades on vähemalt ligikaudu $\frac{1}{32}$ kõigist võimalikest elementidest. Kui hulgad sisaldavad vähem kui $\frac{1}{32}$ elementidest, on efektiivsemaks struktuuriks täisarvude massiiv. Kettale salvestatuna on bitivektorid efektiivsemad kui tihedus on suurem kui $\frac{1}{120}$.

Semestritöö praktilise tulemusena valmis hulgatehete teek Libsets, millesse on koondatud SPEXS-tarkvaras kasutatavad bitivektoritel ja täisarvumassiividel põhinevad hulgaklassid. Teeki saab kaasata paljudesse andmekaeverakendustesse, näiteks sagedaste mustrite otsingusse või assotsiatsiooniuuringutesse. Hulgateegi funktsionaalsuse demonstreerimiseks olen töösse lisanud eksperimentaalse programmi, milles otsitakse teataval viisil järjestatud geenihulkadele ühiseid nimetajaid geeniontoloogiate baasist GO. Lisaks tutvustasin andmekaevesüsteemi, mille ülesandeks on andmebaasist sagedaste alamhulkade kaevandamine ning nende abil teatavate baasis kehtivate reeglite ennustamine.

Töö edasiarendustena tasub kindlasti uurida hulkade realiseerimise võimalusi ja efektiivsust andmebaasiplatvormil. Teiseks oleks huvitav hulkadega eksperimenteerida 64-bitisel arvutiarhitektuuril. Andmekaeveperspektiivist vaadatuna leiab mitmeid uurimisülesandeid näiteks seoses sisuliste geeniuuringutega hulgatehete abil.

Viited

[ABB00] M. Ashburner, C.A. Ball, and J.A. Blake. Gene ontology: tool for the unification of biology. *Nature Genetics*, 25:25–29, 2000.

[FPSM92] W. J. Frawley, G. Piatetsky-Shapiro, and C. J. Matheus. Knowledge discovery in databases - an overview. *Ai Magazine*, 13:57–70, 1992.

[go04] Go: Gene ontology consortsium (www.geneontology.org), 08.05.2004.

[HCI04] M.A. Harris, J. Clark, and A. Ireland. The gene ontology (go) database and informatics resource. *Nucleic Acids Res*, 32 Database issue:D258–61, 2004.

[Juh03] M. Juhkam. Klasterdamine andmekaevanduses. In *Andmekaevandamise uurimisseminar*, pages 70–89, 2003.

[KH95] Martin L. Kersten and Marcel Holsheimer. On the symbiosis of a data mining environment and a DBMS. In *92*, page 12. Centrum voor Wiskunde en Informatica (CWI), ISSN 0169-118X, 30 1995.

[osd04] Operating system technical comparison (www.osdata.com), 09.05.2004.

[VBJ+00] J. Vilo, A. Brazma, I. Jonassen, A. Robinson, and E. Ukkonen. Mining for putative regulatory elements in the yeast genome using gene expression data. In *Proc. of Eighth International Conference on Intelligent Systems for Molecular Bio logy (ISMB-2000)*, volume 8, pages 384–394, La Jolla, California, 2000. AAAI Press.

[Vil02] Jaak Vilo. *Pattern Discovery from Biosequences*. PhD thesis, 2002.

Set arithmetics and applications in Data Mining

Term paper

Jüri Reimand

Abstract

Sets are one of the simplest and yet most important concepts in mathematics and computer science. A great deal of programming tasks has some relations to set theory and many algorithms rely on fast set arithmetics. In Data Mining, set operations are used for finding similar subsets in large amounts of data.

In this term paper we have studied the use of bit vectors and integer arrays as fast structures for integer set realization. We have examined the effectiveness of the two structures with different input data, comparing storage space demands, speed of the input from textual representation as well as the speed of set operations. In conclusion it can be said that integer sets should be implemented as bit vectors if the sets include approximately more than $\frac{1}{32}$ of the elements in the given domain. If the sets are smaller than $\frac{1}{32}$ of the domain's size, integer arrays are generally more efficient. The storage space comparison showed that integer sets should be stored as bit vectors unless the sets include less than about $\frac{1}{120}$ of the elements in domain.

As a practical result of the paper we have gathered the set classes of the pattern discovery tool SPEXS into a C++ library called Libsets. The library can be used in various Data Mining tasks like pattern discovery or association research. In order to demonstrate the functionality of the library was tested on a problem to identify common descriptions to certain gene sets from the GO gene ontology data set. In addition the term paper includes a short overview of a Data Mining system that attempts to predict common rules in a data set by detecting frequent subsets.

A Teegi Libsets kompileerimine

Teegi Libsets loomiseks tuleb esmalt soovitud C++ klassid eriviitadega kompileerida. Viit `-fPIC` (või ka `-fpic`) (ingl. k. *Position Independent Code*) tähendab, et koodi genereerimisel on kasutuses relatiivsed mäluaadressid absoluutaadresside asemel. Viit -c käsib kompilaatoril linkimise etapi vahele jätta.

```
> cc -fPIC -c Sets.C -o Sets.o
```

Teegi saamiseks kompileeritakse saadud objektifailid viidaga `-shared` (mõne kompilaatori korral ka `-g`).

```
> cc -shared error.o Sets.o Set_of_Sets.o
   config.o char_mapping.o config.o -o libsets.so
```

Teegi päisefaili `libsets.h` loomiseks liitsin kokku kõikide teeki kuuluvate failide päisefailid.

```
> cat Sets.h Set_of_Sets.h error.h char_mapping.h
   config.h > libsets.h
```

Programmis (näiteks `Sets_go.o`) teegi kasutamiseks tuleb kompileerida `-L` ja `-lsets` viitadega. `-L` viidale järgneb teegi kataloog. Viit `-lsets` märgistab teegi nimetust ilma '`lib`'-prefiksita ja '`.so`' laiendita (teegifaili nimi on `libsets.so`). Järgnev käsk lingib jooksvast kataloogist '`.`' teegifailid objektifailile `Sets_go.o` ja tulemusena tagastab progammi `Sets_go`.

```
> cc Sets_go.o -L. -lsets -o Sets_go
```

Juhul, kui teek ei ole paigutatud süsteemi teekide kataloogi, peab käivitamise ajaks süsteemimuutujas `LD_LIBRARY_PATH` olema defineeritud kataloog, milles kasutatav teek asub. Muutuja väärtust saab kontrollida käsuga `echo`.

```
echo LD_LIBRARY_PATH
```

Kui muutujat ei ole määratud, saab selle defineerida järgnevalt.

```
> LD_LIBRARY_PATH=/tee/juurest/teegi/kataloogini
> export LD_LIBRARY_PATH
```

Kui muutuja on määratud, on soovitatav teegi asukoht olemasolevale muutujale juurde liita.

```
> LD_LIBRARY_PATH=/tee/juurest/teegi/kataloogini
   :${LD_LIBRARY_PATH}
> export LD_LIBRARY_PATH
```

Päisefail tuleb lisada programmi `Sets_go` päisesse direktiiviga `#include "libsets.h"`. Sellisel juhul peab teegi päisefail asuma programmiga samas kataloogis.

Teegi hõlpsamaks kompileerimiseks olen lisanud kompileerimismakro `Makefile`, mida saab kasutada käsu `make` abil. Lisaviidaga `make prog` käivitades kompileeritakse vaid programmifail (praegusel juhul Sets_go). Lisaviit `make clean` kustutab kõik kompileeritud kahendfailid ning `make cleanprog` ainult programmi kahendfaili.

B Programmid

Selles lisas on toodud kaasatud programmide lühijuhendid.

B.1 Sets_go

Asukoht: `libsets/Sets_go`, `libsets/Sets_go.C`

Programmile antakse käsureaparameetritena SPEXS-formaadis hulgafailid (1) geenidest ja (2) GO funktsioonidest. Töö eelduseks on mõlema faili olemasolu. Hulgafailide esimesel real peab olema `<D:nr>`, kus 'nr' on domeenisuurus. Teisel real peab olema `<N:nr>`, kus 'nr' on hulkade arv.

Programm trükib standardväljundisse iga geenifaili hulga korral reanumbri ja hulga summaarse ülekatte üle kõigi GO kategooriate.

Lisaparameetrid:

`V` - 'verbose mode', lisaks ülalmainitud infole kuvab programm kõikvõimalikku muud infot, trükib välja hulgad, jms.

`C` - 'convert mode', enne operatsioonide sooritamist teisendatakse bitivektorid massiivideks või vastupidi.

B.2 Sets_input

Asukoht: `libsets/Sets_input`, `libsets/Sets_input.C`

Programmile antakse käsureaparameetritena SPEXS-formaadis hulgafail. Töö eelduseks on faili olemasolu. Hulgafaili esimesel real peab olema `<D:nr>`, kus 'nr' on domeenisuurus. Teisel real peab olema `<N:nr>`, kus 'nr' on hulkade arv.

Programm trükib standardväljundisse faili sisselugemiseks kulunud aja sekundites.

Lisaparameetrid:

`V` - 'verbose mode', lisaks ülalmainitud infole kuvab programm kõikvõimalikku muud infot, trükib välja hulgad, jms.

`C` - 'convert mode', enne operatsioonide sooritamist teisendatakse bitivektorid massiivideks või vastupidi.

B.3 Sets_prefer

Asukoht: `libsets/Sets_prefer`, `libsets/Sets_prefer.C`

Programmile antakse käsureaparameetrina SPEXS-formaadis hulgafail(1) ning eelistuskoefitsent$[0..1]$. Eelistuskoefitsendi kohaselt teisendatakse massiive bitivektoreiks, kui $domeeni\_suurus * koefitsent > hulga\_suurus$, ja bitivektoreid massiivideks vastasel korral. Hulgafaili esimesel real peab olema `<D:nr>`, kus 'nr' on domeenisuurus. Teisel real peab olema `<N:nr>`, kus 'nr' on hulkade arv.

Programm trükib standardväljundisse hulkade operatsioonideks (vahe, ühisosa) kulunud aja. Teisendusaega ei arvestata.

Lisaparameetrid:

`V` - 'verbose mode', lisaks ülalmainitud infole kuvab programm kõikvõimalikku muud infot, trükib välja hulgad, jms.

C Libsets hulgaoperatsioonid

Järgnevas on toodud Libsets teegi hulgaoperatsioonid bitivektorite `BA_Sets`, massiivide `L_Sets` ning mähisklassi `Sets` jaoks. Sümboliga (+) on tähistatud realiseeritud funktsionaalsus, (-) tähistab puuduvat funktsiooni ning (X) näitab, et funktsioon tagastab vea `not implemented` või töötab vaid ühte tüüpi hulgaga.

Funktsioon	BA_Sets	L_Sets	Sets
`void del_set(i_set S)` Hulga kustutamine	+	+	+
`int next(i_set set, int& it)` Hulgast järgmise elemendi lugemine	+	+	+
`i_set create()` Tühja hulga loomine	+	+	+
`i_set create(int Size)` Etteantud suurusega tühja hulga loomine	-	+	+
`i_set create(char* STR)` Hulga lugemine kujul (-)|S|<X:,e1,e2..>, X on B või L vastavalt tüübile	+	+	+
`int compute_size(i_set S)` Hulga suuruse arvutamine, esimesel kohal oleva väärtuse kontroll	+	+	+

`void print`
`(i_set S, ostream& OS , int type)`
Hulga väljund kujul (-)|S|<X:,e1,e2..>,
X on B või L vastavalt tüübile
	+	+	+

`int lookup(i_set S, int elem)` Hulgast elemendi otsimine	+	+	+

Funktsioon	BA_Sets	L_Sets	Sets

`int intersect`
`(i_set A, i_set B, i_set Result)`
Hulkade ühisosa leidmine
Hulk `Result` peab olema loodud, see kirjutatakse üle
	+	+	+

`int set_union`
`(i_set A, i_set B, i_set Result)`
Hulkade ühendi leidmine
Hulk `Result` peab olema loodud, see kirjutatakse üle
	+	X	X

`int substract`
`(i_set A, i_set B, i_set Result)`
Hulkade vahe leidmine
Hulk `Result` peab olema loodud, see kirjutatakse üle
	+	+	+

`int complement`
`(i_set A, i_set B, i_set Result)`
Hulga täiendi leidmine
Hulk `Result` peab olema loodud, see kirjutatakse üle
	+	X	X

`int equal(i_set A, i_set B)` Hulkade võrdsuse testimine	+	+	+
`void remove (i_set S, int n)` Hulgast elemendi eemaldamine	+	X	X
`void insert(i_set S, int n)` Hulka elemendi sisestamine	+	X	X
`int size (i_set S)` Hulga suuruse lugemine	+	+	+

Funktsioon	Sets
`int Prefer_L(int Size)` Funktsioon soovitab eelistada massiivi, kui tihedus den(S) < 1/32	+
`I_set Make_L_from_BA(i_set BAS)` Massiivi teisendamine bitivektoriks	+
`I_set Make_BA_from_L(i_set S)` Bitivektori teisendamine massiiviks	+
`I_set make_preferred(i_set S)` Funktsioon loob hulga vastavalt eelistusele	+
`I_set min_copy(i_set S)` Koopia antud hulgast vastavalt eelisele	+

`Void`
`Minimise_representation_in_same_area`
`(i_set S)`
Antud hulk kirjutatakse üle vastavalt eelisele
	+

`int cmp(i_set A, i_set B)` Sama tüüpi hulkade võrdlus	+
`int BA_L_equal(i_set bas, i_set lis)` Eri tüüpi hulkade võrdlus	+
`int gcmp(i_set A, i_set B)` Üldistav võrdlusfunktsioon	+
`i_set copy( i_set S )` Hulga kopeerimine	+
`int Is_BA( i_set S )` Hulga tüübi uurimine	+

D Failide loetelu

Järgnevas on toodud kaasasolevate failide loetelu.

./libsets

char_mapping.C

char_mapping.h

config.C

config.h

error.C

error.h

infrastructure.c

infrastructure.h

libsets.h

libsets.so

Makefile

Set_of_Sets.C

Set_of_Sets.h

Sets.C

Sets_go

Sets_go.C

Sets.h

Sets_input

Sets_input.C

Sets_prefer

Sets_prefer.C

./setgen

setgen.pl

Setgen.pm

./setrand

setrand.pl

Setrand.pm